VTScada

Software for Monitoring & Control

Programmer's Guide

Programmer's Guide

Excerpts from the VTScada Help Files

Printed in Canada

Trihedral Engineering Limited

Head Office
1160 Bedford Highway, Suite 400
Bedford, Nova Scotia
Canada
B4A 1C1

Phone: 902-835-1575
Toll free: 800-463-2783
support@trihedral.com
sales@trihedral.com

Trihedral UK Limited

Glover Pavilion, Campus 3
Aberdeen Science Park
Balgownie Drive, Aberdeen
UK, AB22 8GW

Phone:+44 (0) 1224 258910
Fax: +44 (0) 1224 258911

Trihedral Inc.

Suite 160
7380 Sand Lake Road
Orlando, Florida
USA
32819

Phone: 407-888-8203
Fax: 407-888-8213

Trihedral Calgary Office

Suite 505 - 888 4 Ave SW,
Calgary, AB,
T2P 0V2
403.921.5199

Contents

Scripting and Automation

Intended audience: Advanced developers who want to create new features for their application.

VTScada includes (and is largely built with) its own programming language. You can use this language to create unique tools for your application development work, including custom tags, script applications, wizards and more. Support is provided for all the features you would expect in a programming language, and for many features that are unique to VTScada.

Related Information:

...Start Here for Scripting and Automation - For those who are new to VTS scripting. Covers the fundamentals and helps you get started with tag-based expressions.

...The VTScada API - The complete introduction to the VTS language

...Basic Programming Tasks - Examples of how to achieve common goals.

...Custom Tag Types - Creating new types of tags from scratch.

...Programming Parent Tags - Included for historical reference only. Obsolete.

...The VTScada Wizard Engine - Create user-interface wizards for customization tasks.

...Communication Drivers - How to write a driver.

...Programming Other Modes of Communication - Alternative I/O options.

...Cryptography in VTS- Guide to cryptography functions that you can use.

...Web Services and XML - How to use SOAP services to create an automated interface to VTScada.

...- Writing your own WAP modules.

...Alarm Manager – Guide to the features and tools in the alarm manager.

...Historian – API and Queries – Guide to the features and tools in the historian manager.

...Modem Manager Service – Guide to the features and tools in the modem manager.

...Socket Server Manager – Guide to the features and tools in the socket server manager.

...Security Manager – Guide to the publicly accessible tools in the security manager.

...Time Synchronization Manager Service – An overview of the time sync. service.

...RPC Manager Service – Detailed guide to remote procedure calls.

...Debugging and Analysis – Instructions for using the various tools included with VTScada.

...Function Reference – See the VTScada Function Reference.

...Application Properties – See the VTScada Admin Guide. Control of your application's appearance and behavior through settings.

Start Here for Scripting and Automation

If you are exploring the scripting and automation tools in VTScada, it is likely that you will use some combination of the following components to reach your goal.

On schedule, or in response to conditions...

- A tag's value changes or reaches a set point.
- A set time arrives, or an interval of time passes.
- An operator acts, including button-presses and security events.

... perform a calculation...

- Calculate values or words.
- Store a value.

... and do a task

- Display a message.
- Run a report.
- Set or write a value.
- etc.

Before starting to write script code, you must decide where to place it. You have a choice between three options:

- Expressions are written inside tags. The expression uses a sub-set of the scripting language to calculate a new value, and relies on the tag to save or write that value. Choose this option for most tag-related automation.
- A module can make use of the entire scripting language and is stored in a file, located in your application folder. How it is used is determined by how it is declared, whether as a tag type, a **service**[1], a report, etc. Choose this option for reports, custom tag types that cannot be built in the Tag Browser, enhanced security monitoring, etc.

---

[1] A VTScada module that is continuously running, usually waiting for an event to trigger some action.

- A Script Tag combines the two, linking a module to a tag. The linked tag is used either as a data source, a trigger, or both. Choose this option for tag-related automation that requires tools from the entire VTScada scripting language.

The following list of topics will provide everything you need to get started with scripting and automation:

Related Information:

Where to write your code...

...Expressions in Tags and in Widget Properties

...Script Code in Modules

...Configuring a Script Tag

On schedule, or in response to conditions...

...Test Conditions - Compare values and select which block of code to run.

...Access a Tag Value or Application Property - Tools for accessing application data.

...Mark the Passage of Time - Tools to run your code on time.

...Obtaining User Input - Tools to watch for buttons and key-presses.

Perform a calculation and do a task.

...Math Functions in Expressions

...Text Functions in Expressions

...Time and Date in Expressions

...Examples of Expressions

...The VTScada API - Complete reference to the VTScada scripting language.

...Basic Programming Tasks - How to build common module types.

Quick Reference Guide for Expressions

This topic is for people who have some programming experience and only want to see how fundamental tasks are done in the VTScada expression editor. Other topics in this chapter provide a comprehensive reference. You may also find the topic Expression Examples useful.

Reference Tag Values:

Tag values are referenced in expressions as follows:

```
[tag name]
```

Tag name must be the actual tag name string. This is a robust form to use since it accommodates tag names with spaces and other punctuation symbols.

Example: Averaging two tag values…

([Pump1 FlowRate] + [Pump2 FlowRate]) / 2

Reference Other Tag Parameters:

You can access parameters other than Value within tags. A few common ones such as ScaledMin and ScaledMax might be useful to your expressions.

Format:

```
[Tag Name]\ParameterName
```

Parameters will vary from one tag type to another and there is no guarantee that they will not change in future versions. Nonetheless, some common parameter names are as follows:

- All tags:
  \Value
  \Name
  \Area
  \Description
  \Questionable
- Analog tags:
  \ScanInterval
  \UnscaledMin & \UnscaledMax

\ScaledMin & \ScaledMax
\Units

- Digital Input tags:
  \Bit0Address
  \Bit1Address
- Digital Output tags:
  \Address
  \DataSourceTag

You can use the Source Debugger application to discover what variables are available within any particular tag in your version of VTScada.

Comments

Comments are always useful and can be located anywhere in an expression. They are enclosed in braces: {This is a comment }

Substitute a default if a tag's value is INVALID

A communication error or other event may result in an expression breaking when a tag's value goes to INVALID. You can substitute a default when this happens by using the PickValid function. This has the general form: PickValid(expr_1, optional_expr_2…, some_guaranteed_constant). For example, the following will return a zero if the flow rate Analog Input tag is invalid:

```
PickValid([Pump2 FlowRate]), 0)
```

Constants

Constant values include text and numbers (integers, floating point, binary and hex). These can be part of an expression as follows: Concat ("Level is ",[TankLevel_1]) or [TankLevel_1] / 3.
The value returned by an expression is whatever was last calculated. If the expression contains nothing but a constant, then that is what is returned. This is useful for displaying one of two messages depending on current conditions (see the example for the If Else Operator)

Basic Math Operators

- + Addition
- - Subtraction
- * Multiplication

- / Division

Use parenthesis to control the precedence of operations:

```
(2 + 3) / 4
```

Value Comparisons & Relational Operators

- < Less than
- <= Less than or equal to
- == Equivalent
- => Greater than or equal to
- > Greater than

Caution! Note that the test for equivalency is *two* equal signs, not one. Here is an example of good code:

```
[ValveFlow] == 1 ? "On" : "Off"
```

The following is an example of a very serious programming error:

```
[ValveFlow] = 1 ? "On" : "Off"
```

Boolean Comparisons:

- && And
- || Or

If-Else Operator

- ... ? ... : ...

  test expression ? expression if test is true : expression if test is false

Example: Return a warning message if a tank level exceeds 90...

```
[TankLevel] > 90 ? "Dangerous level!" : "Level is safe"
```

Note that you can cascade these by adding new If Else operators, instead of the simple text constants shown in this example.

Cond Function

The Conditional function does the same thing as the ?: (If Else) operator. The only difference is in how it is written: Cond(test_expression, expression_for_true_case, expression_for_false_case)

```
Cond([TankLevel] > 90, "Dangerous level!", "Level is safe")
```

Mathematic functions

Sqrt - Square root

Log - Logarithm
Min - Minimum of a list of values
etc. - see: Mathematic Functions

Read application property values

To access a value from the Settings.Dynamic file, you can use the following general form:

```
\Code\VarName
```

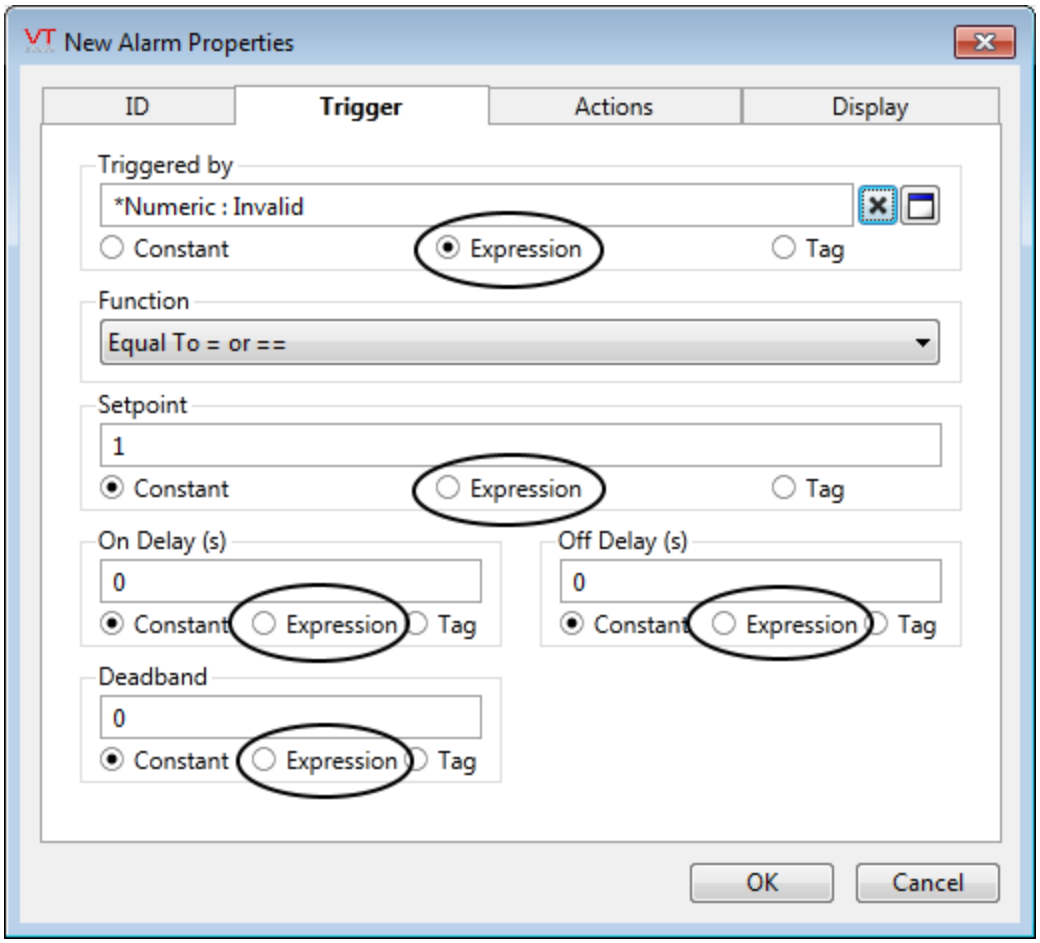
Expressions in Tags and in Widget Properties

An expression is "any calculation that returns a result".
In more practical terms, an expression is something that...

- Can combine or compare multiple tag values to better monitor your system.
- Can signal a need for control actions, based on any set of system conditions.
- Can take into account the time, date, logged-in operator, system status, etc.
- Can extend the capabilities of VTScada to meet any of your SCADA needs.

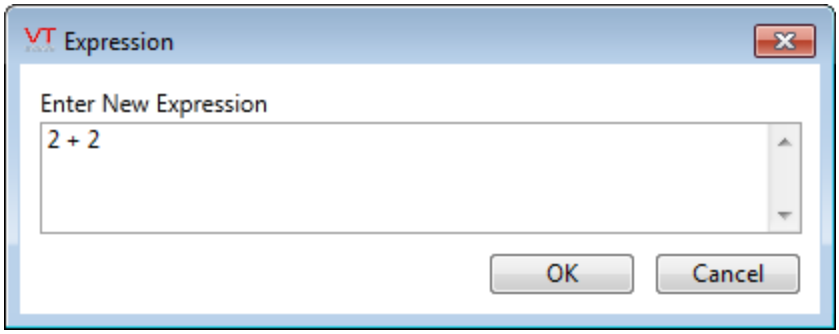
General steps to create an expression:

In any VTScada tag configuration field that has the options, Constant, Expression, Tag...



1. Click the Expression option to select it.
2. Click the expression editor button.

3. Enter an expression into the editor window.



The expression editor window can be re-sized if required for a longer expression.

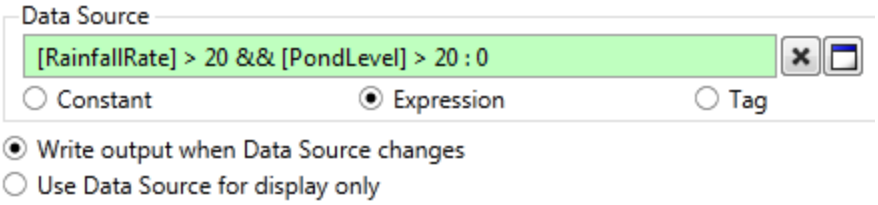
4. Click OK to save your work and return to the tag configuration.

Note: You cannot save an expression that contains a syntax error. For example: unbalanced parenthesis "2 + (2/3" , or using an operator without an operand "2 + ".

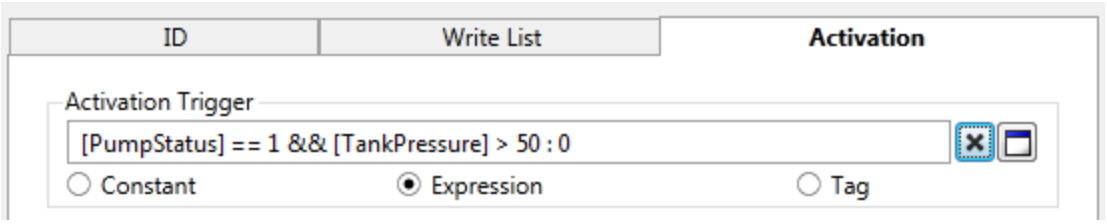
Where to use expressions:

While expressions are most commonly used in Calculation tags, they can also be a data source option for a wide variety of other uses. For example, the title of a parametrized page, or...

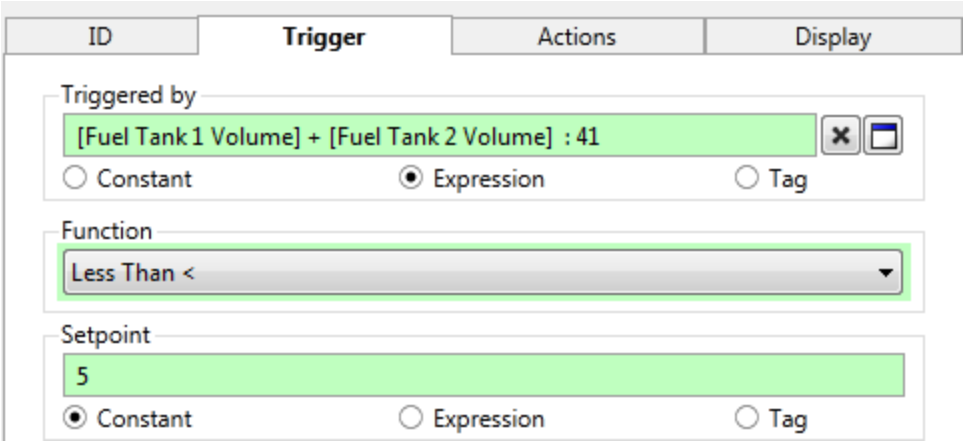
- Analog Control, I/O tab configuration:



- MultiWrite tag, activation tab configuration:



- Alarm tag, trigger tab configuration:



Note: When looking through the reference guide for a function to use in an expression, note the *usage* line of the definition. Many functions are restricted to certain usages.

Related Information:

...Syntax Rules and the Expression Editor

...Quick Reference Guide for Expressions

...Access a Tag Value or Application Property

...Examples of Expressions

Related Functions:

...Math Functions in Expressions

...Comparing Values in Expressions

...Time and Date in Expressions

...Text Functions in Expressions

...Triggers and Events in Expressions

Script Code in Modules

Much of what you see in VTScada was written using the VTScada scripting language. You can leverage the power of this language to write new functions, reports, services, user-interface features and much more.
Like any programming language, there is a great deal to learn. You should start with simple tasks and build your knowledge as you work towards more powerful features. Please begin with the following topics:

Related Information:

...The VTScada API

...Basic Programming Tasks

...Create a New Script Application

Configuring a Script Tag

Script tags are one of the standard types built into VTScada. The script tag links a VTScada script code module to another tag's value. It is used to go beyond the power of in-tag expressions to allow the use of the full VTScada scripting language inside a tag.
The module for a script tag must have two parameters, as follows:

```
(
 PointObj { object value of AI to monitor };
 ScriptObj { Script Object };
)
```

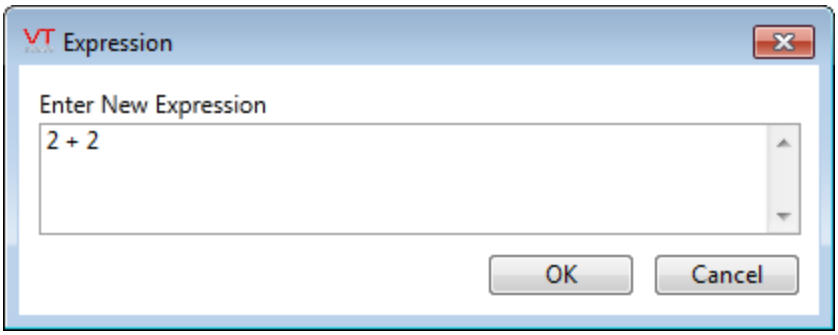
PointObj is the tag whose value is monitored and used by the Script tag. This matches the first field, "In Tag Scope", found in the Execute tab of the Script tag's configuration panel.
ScriptObj is used internally to link to your script tag. Whatever calculation is performed in your module must assign a value to ScriptObj\value. This enables the module to pass its calculated value back to your Script tag.

Related Information:

...Script Tags - Example and details for creating a Script Tag can be found in the VTScada Developer's Guide

Syntax Rules and the Expression Editor

Expressions are written inside an expression editor. They are not typed directly into a tag configuration field. To open the editor, select the expression option below a field, then click the box as indicated.



An expression can be as simple as 1 + 1, or even just "1". Expressions can be as simple or complex as you need, but must always return exactly one value. An expression that uses a variety of calculations on a multitude of tags is perfectly acceptable, so long as the final result is to find one value.

For example, the following expression is legal. (Multiple calculations on multiple lines, returning one result. Counts pump starts.)

```
Latch(PickValid(Edge([Pump1], 1), 0), Watch(0,
[PumpCounter]))
? (Value + 1)
: PickValid([PumpCounter], 0)
```

The next example is not legal (calculates two separate results).

```
1 + 1
2 + 2
```

If a valid expression has been entered, then when you close the editor you will see the expression displayed, followed by a colon and the calculated result of the expression. You must click on the expression editor's OK button in order to see the result of your expression – there is no preview for the calculation.

An expression result

Note: Expressions are always examined by VTScada for syntax errors before they are saved. If the expression doesn't follow the rules, then you will be notified that a problem exists. Incorrect expressions are not saved.

VTScada follows syntax rules that are common to most programming languages. The following is a quick overview for those who have not studied programming:

- Every opening parenthesis must be matched by a closing parenthesis. (2 + 2)
- Text must be enclosed in quotation marks. Text not enclosed in quotation marks is taken as the name of a variable. X = 2 + 3.
- Every opening quotation mark must be matched by a closing quotation mark. "Hello World"
- To display a quotation mark in text, use a doubled set of quotation marks: "The computer said, ""Hello World"". "
- *Operators* are symbols such as plus and minus signs. *Operands* are the things being operated on by the operators. In the expression 2 + 2, the digits "2" are operands and the "+" is an operator.
    - You should always put a space between each operator and operand. While not strictly required, the spaces will help you avoid errors and will increase the clarity of your code. Operands must always be separated by spaces.
    - Extra spaces are ignored.
    - Line breaks are ignored, except that they count as a space between operators and operands.
- There is a precedence to the order in which mathematic operations are performed (multiplication before addition), but you should use parenthesis to improve clarity and to explicitly control the order of the operations. ((2 + 3) * 5). Evaluation proceeds from the inner-most sets of parenthesis to the outer-most, thus in this example the 2 and 3 are added together before the result is multiplied by 5.

Test Conditions

"If it is raining, then wear rubber boots, otherwise wear shoes."
Conditionals are used to compare values and then direct which actions to take based on the result.
A conditional is built using the following parts:

- A keyword or symbol that indicates "this is a conditional". In VTScada, this takes the form "IfElse", "IfThen", "Cond" or the in-line, " ? : " syntax.
- A test that can evaluate to TRUE or FALSE. This may be a comparison "A >= B", a function that watches the clock TimeArrived(x)", or it can simply be "A" if the value of A can change between zero (FALSE) and non-zero (TRUE).
- The code to run, depending on the result of the test.

In an expression, you may use only IfElse and the in-line ?: conditional.
In a module, you may use these plus IfThen, IfOne, IF, and Cond.
In computer code, there is a special value called "Invalid". It is the value assigned to a variable when no other value can be calculated - for example, x divided by zero. Invalid is not zero or non-zero, it is neither TRUE nor FALSE. All comparisons to Invalid return Invalid.
To deal with Invalid, you have two tools: PickValid() and Valid(). PickValid () takes a list of values and returns the first that is valid. Thus, if you are comparing the value of X, and there is a chance that X may be invalid, you can substitute a known value such as zero using the expression, Pick-Valid(X, 0), thus ensuring that a valid value is always used in the com-parison.
Valid(X) tests whether X is valid or invalid, returning TRUE or FALSE.

Related Functions:

... IF

... IfElse

... IfThen

... IfOne

... Cond

...Inline If-Else.

Related Information:

...Triggers and Events in Expressions

...Comparing Values in Expressions

... Boolean Logic Operators

Comparing Values in Expressions

Any comparison will return 1 as the value of the expression if the comparison is true and 0 if it is not.
The following operators are provided to allow you to compare one value to another value in an expression.

Symbol	Meaning
>	Greater than
<	Less than
==	Equivalent (Notice the double equal sign. A single "=" won't do)
!=	Not Equivalent
>=	Greater than or equal
<=	Less than or equal

An example of a comparison expression is as follows:

```
[Tank1Level] > 50
```

The preceding is a complete expression. The word "IF" is not required, and indeed is not allowed.
Simply returning 1 if a comparison is true and 0 if it is false is useful (a trigger for alarms is one example), but you can also provide your own values to be returned instead. This is done with an If... Else in four steps, as follows:

1. Put a question mark after the comparison to indicate that you have finished the IF part of the expression
2. Put the value or expression you want returned if the expression is TRUE
3. Put a colon to indicate the beginning of the ELSE part.
4. Put the value or expression that you want returned if the comparison is FALSE.

For example, the following expression will display the words "Within safe limits" while a tank's level is below 50%, but will display "Above safe limits" when the value rises above 50:

```
[TankLevel_1] <= 50 ?
        "within safe limits" :
        "above safe limits"
```

You are not restricted to constants for the values that are to be returned. Any valid expression may be used for each of the two cases.
You could write the same expression using the IfElse function as follows:

```
IfElse([TankLevel_1] <= 50, "within safe limits", "above safe limits")
```

Multiple Comparisons

Often, you will want to check more than just one tag. For example, if a tank level is above 90% AND the safety valve is closed, then open the valve. (If the valve is already open, there's no point opening it, and if the tank level is below 90% you also don't want to open the safety valve.)
You can join as many comparisons as you want together with the symbols && (which means "AND") and || (which means "OR"). Don't forget that you can use parenthesis to make it clear which value is being compared to which.
The following example will return true only when both the level in Tank 1 and the level in Tank 2 exceed 80%

```
[TankLevel_1] > 80 && [TankLevel_2] > 80
```

The next example returns true whenever either tank exceeds 80%

```
[TankLevel_1] > 80 || [TankLevel_2] > 80
```

Note: Remember! Use a double equal sign for comparisons "==". A single one means "assign" not "compare".

Triggers and Events in Expressions

There are many functions that can be used to watch for a triggering event that will start your code running.
When used in a tag-based expression, these functions will remain latched after they have been triggered. Once the time has gone by, the

value has been reached, or the key has been pressed, your code will run, but will not run again the next time. To reset the trigger for the next event, you will need the Latch() function in your tag-based expressions (description follows).
In your module-based script code, this does not apply. There is still a use for the Latch() function, but many functions will automatically reset themselves. This is noted in each function description.

Example:
The goal is to create an expression that toggles from TRUE to FALSE and back again, switching every second. So, every two, four, six, eight ... seconds, it becomes TRUE and every 3, 5, 7, 9 ... seconds it becomes false.
AbsTime(1, 2, 0) will switch to TRUE on every multiple of a two-second interval. But, in a tag-based expression, once it triggers to TRUE, it will simply stay there. Let's add a Latch() to take care of this.
The Latch() function takes two parameters: when the first becomes TRUE, the Latch() returns TRUE and the second parameter is reset if it was TRUE. When the second becomes TRUE, the Latch () returns FALSE and the first parameter is reset if it was TRUE.
The answer to the problem is:

```
Latch(AbsTime(1, 2, 0), AbsTime(1, 2, 1))
```

Both AbsTime functions cycle every two seconds, but the second is offset by one second. The first parameter switches the Latch on every two seconds. The second switches the Latch off one second later, and resets the first parameter so that it can switch back on.

Related Functions:

... Latch

...Latching and Resetting Functions

Related Information:

...Comparing Values in Expressions

...Access a Tag Value or Application Property

...Mark the Passage of Time

...Obtaining User Input

Access a Tag Value or Application Property

When writing an expression within a tag, you can obtain and use the value of any tag in your application using the following syntax:

```
[TagName]
```

This creates an absolute reference to the tag matching that name. If you have a hierarchical tag structure, then you should provide the full name to the tag. Relative paths can also be built using dots and backslashes. For example, to refer to a tag two levels up in the hierarchy, use:

```
[..\..\TagName]
```

The "value" property is assumed by default. Any other property of the tag may be accessed by adding "\PropertyName" following the [TagName].

```
[TagName]\Area
[TagName]\ScaledMin
```

Property Values

Also within a tag expression, application properties can be read as follows:

```
\Code\PropertyName
```

For example, if you want to find the total value of three analog inputs that are monitoring tank volumes, you might write the following expression:

```
[TankVolume_1] + [TankVolume _2] + [TankVolume _3]
```

Note: Caution: Tag values may be "Invalid", especially at system startup. The result of any calculation on an Invalid is always Invalid. (For example, 1 + 0 is 1, but 1 + Invalid is Invalid.)

To avoid errors and the possible repercussions of returning an unwanted Invalid value, you can use the PickValid function to supply a default

value. The PickValid function examines a list of values that you provide as parameters. The first value that is not Invalid will be the one returned.

```
PickValid( [TankVolume_1], 0)
```

- Returns the value of TankVolume_1 when that tag's value is Valid
- Returns 0 when the tag's value is Invalid.

The expression shown earlier that calculates the sum of three tank volumes can be written as follows:

```
PickValid([TankVolume_1], 0) +
PickValid([TankVolume _2], 0) +
PickValid([TankVolume _3], 0)
```

Should any tag's value go to Invalid for any reason, this expression will substitute 0 for its value.

Note that the PickValid function should be used with care: There may be instances where it would be better to return an invalid rather than an incorrect value.

If you are writing script in a module, rather than an expression, use the following syntax to refer to a tag's value:

```
Scope(\VTSDB, "FullTagName")\Value
```

It is often worth the extra effort to find and use the tag's unique ID value rather than its name. This ensures that your code will continue to work after a developer moves or renames the tag.

```
Scope(\VTSDB, "... tag's unique ID value...")\Value
```

Related Information:

...Relative Tag and Property References - addressing parent-child tag structures

Relative Tag and Property References

The tag that you are referencing in your expression may not be at the same level in a parent-child tag structure as the tag that contains the expression. There are several ways that you can specify the relationship between tags.

Note: In older versions of VTScada, the standard method of referencing a tag was to use the expression, "Variable("TagName")\value". You may type that if you wish, but upon saving the expression, it will be replaced with the most appropriate of the following for the situation.

Code	References...
[TagName]	Open Relative Path. References the closest tag with that name.
[ParentName\TagName]	Finds the closest tag with this match. Common ancestors above ParentName need not be included
[*ContextName\ChildName	Ancestor Relative Path. References a sibling tag's value, within a user-defined type. This format works across multiple instances of the type, always finding the sibling tag in the local instance.
[..\..\TagName]	Fixed Depth Path. Finds the tag having that name, at distance above the current tag, as specified by the number of .. repetitions.
[<\Full Path\Tag Name>]	Absolute path. Equivalent to providing the GUID of the tag in question. This will continue to refer to the same tag, even if that tag is moved or otherwise renamed. The full path must be provided so that the correct tag can be found. If that tag is moved, the path will update automatically. This form of address is seldom used in a user-defined type since it refers to one specific tag and is not relative to each instance of the type.
[*TagType]	References the nearest ancestor of the specified tag type. Valid options include: *Port (nearest port tag ancestor), *Device (nearest driver tag ancestor), *Trigger (nearest Trigger tag ancestor), *Numeric (nearest numeric tag ancestor), and *SQLLoggerGroup. If the current tag is of the same type as the one being referenced (common with *Numeric) then it is necessary to explicitly point away from the current tag by adding "..\". Use this to create expressions that can be transferred from one parent to another, and will automatically find the nearest appropriate parent tag. For example: [*Driver].

	Assuming there are several stations, each with its own parent driver tag, this expression can be taken from one station to ensure that I/O tags are linked to the appropriate driver.
PropertyName	Uses the value of that property, as defined in the current tag.
..\PropertyName	Uses the value of that property, as found in the parent tag. Will automatically continue searching upwards through the tree for an ancestor that has this property.

Mark the Passage of Time

The functions listed here can help you write code that will execute on schedule. Developers who are creating tag-based expressions should note that the Trigger tag is designed for exactly this purpose, and in certain cases may be easier to use than an expression.

AbsTime(Enable, Interval, Offset)	When enabled, becomes true when any multiple of Interval seconds since midnight has passed. Times that are not evenly divisible by Interval can be obtained by adding an offset.
CurrentTime(Type)	Returns the number of seconds since Jan 1, 1970. Set Type to 2 for UTC time.
TimeArrived(UTCTime)	Becomes true when the specified time (in UTC format) arrives.
RTimeOut(Enable, NumSeconds)	A cumulative timer. Becomes true when NumSeconds have passed.
TimeOut(Enable, NumSeconds)	A continuous timer. Becomes true when an uninterrupted NumSeconds have passed.

Related Functions:

...AbsTime

... CurrentTime

... TimeArrived

... TimeOut

... RTimeOut

Obtaining User Input

The tools listed in the topics of this chapter will help you watch for user input, whether to request information or to respond to control events. Choose the appropriate tool for the task: for much application development work, it is far easier to use the widgets designed for control and output tags than to write a user interface from scratch.
For programmers who are building dialog boxes for tag configuration: In addition to the tools listed in this chapter, VTScada provides a set of tools designed for use in tag configuration dialogs. These are the so-called, "P-Tools," where "P" refers to "parameter editing".

Note: Tab order between user input controls follows their z-order (that is, the order of the statements within the state), rather than their Focus ID value.

Related Information:

...Mouse Input

...Keyboard Input

...Selection Input

...Placing Focus on an Object vs. Selecting an Object

Mouse Input

Several VTScada functions are designed to accept input from the user via the screen pointer (mouse). Some, such as Pick, and ZButton are designed to watch for click actions. Others, such as Target, XLoc and YLoc simply watch the location of the pointer, and can be used to trigger an action when the operator moves the pointer to a defined area.

All of the GUI-graphics commands (GUIButton, GUIBitmap, etc.) are designed to watch for mouse input, and will return a numeric value indicating which combination of mouse buttons were used when an operator clicks on the graphic.
To use these functions, they must be placed in a module that is contained in a window. They can be used as the trigger condition of an action to allow scripts to be executed when an operator clicks within a specified area.

Example:

```
If Target(120, 50, 220, 80);
[
   ...
]
```

This statement will cause the script to execute whenever the mouse passes over the target area.

Related Information:

...Keyboard Input

Related Functions:

... Pick

...Click

... ZButton

... GUIButton

... Target

... XLoc

... YLoc

Keyboard Input

Keyboard input may be used to allow an operator to respond Y (Yes) or N (No) in response to a prompt. It is also used whenever there is a need to prompt for a numeric or text value.
In most operating VTS applications, tags that require user input are drawn using an appropriate, built-in widget. The tools discussed here

are used in script applications, which do not have access to the tag widgets, or in a heavily customize page of an application. If designing a configuration dialog for a new type of tag, you should use the various P-Tools, which have been designed specifically for use in configuration dialogs.
The most commonly used keyboard functions are MatchKeys and ZEditField. MatchKeys is typically used to trigger an action when a specified key or sequence of keys is pressed. It does not display the keystrokes entered.
The ZEditField function is used to accept text entered by the operator. Before typing, the operator must activate the object by clicking on the area of the window where the graphic is displayed, or by pressing the TAB key to set the focus to that input object.

Related Information:

...Mouse Input

...ASCII Constants

Related Functions:

... Keys

... MatchKeys

... ZEditField

Selection Input

Radio boxes, drop lists, combo-controls and check boxes are all examples of selection input. For all of these tools, the user is provided with a selection of options from which they can choose. Also in this group are tools such as the VTScada color selector.

Related Functions:

... CheckBox

... ColorSelect

... Droplist

... Listbox

... RadioButtons

... Spinbox

... WinComboCtrl

Usage Rules for Functions

VTScada code runs in two modes: Script or Steady State. Many functions will work in only one mode. The "Usage" line in each function description tells you the mode where the function can be used.

Note: Just because a function can be used in a given situation, doesn't mean that it should be. For example:
* It makes no sense to put a graphics function into a Calculation tag's expression.
* MatchKeys will capture keystrokes only when used in a window or page, not in a service or Calculation tag.
* Script-mode functions can be used for optimized tag parameter configuration, but many are not appropriate in that context.

If you are writing...

General Expressions (Calc. tags)
If you are writing an expression for a Calculation tag, or anywhere that you have the option "Constant / Expression / Tag":

If the function is marked as "Script Only" then you cannot use it here.
If the function works in Steady State, then it will compile when used in a Calc tag expression, but it may or may not be useful there. For example,

Tag Parameter Expressions - Optimized
Only functions that can be used in Script may be used for optimized tag parameter expressions. These expressions are evaluated as the tag is

initialized, then not run again during normal operations. You cannot use Steady State-only functions in this situation.

Tag Parameter Expressions - Not Optimized

Only functions that can be used in Steady State may be used for non-optimized tag parameter expressions. These expressions are re-evaluated whenever any of the parameter values change. You cannot use Script-only functions in this situation.

Page Code, Services, Reports, etc.

These are full VTScada modules, declared in the application's AppRoot file. The full VTScada language and function list can be used.

Math Functions in Expressions

The symbols on your keyboard to use for the basic math functions are as follows:

- `+` addition
- `-` subtraction
- `*` multiplication
- `/` division

There are rules of precedence to control which operations are done first. For example, multiplication and division happen before addition and subtraction. You may wish to use parentheses in order to override the rules or to make your intentions clear.

```
4 + 3 * 2 : 10
(4 + 3) * 2 : 14
```

There is no limit to how many sets of nested or consecutive parentheses you can use. Just be sure that for every opening parenthesis, there is a matching one to close.

The following are a few of the mathematic functions available to your expressions. For a complete list, see Math functions in the VTScada Function Reference. (All math functions can be used in an expression.)

Max(X, Y, Z, …)	Returns the variable having the largest value.
Pow(X, Y)	Returns the value of X raised to the power of Y.
Sqrt(X)	Returns the square root of the value in X.
Int(X)	Returns the number with any digits following the decimal point truncated.
Sin(X)	Returns the trigonometric sine of X.
Cos(X)	Returns the trigonometric cosine of X.

Related Functions:

...Math - Generic

...Math - Rounding

...Math - Trigonometric

Text Functions in Expressions

Expressions can be used to display calculated text as well as numeric values. For example, you might use the Concat() function to join the value of a tag or the result of a calculation to a sentence.
A few of the string handling functions in VTScada are as follows. See String and Buffer Functions for complete descriptions of these and other functions. (Note: many string functions cannot be used in Steady State, and thus cannot be used in an expression.)

Concat(a, b, c…)	Concatenates any number of sub-strings into one sentence.
	`Concat("Level of Tank 1: ",[TankLevel_1], "%")`
	Returns (for example): "Level of Tank 1: 30.35552%"
	`Concat(" Viewing Station: ", StationNumber)`
	Example shows how to set the title of a parametrized page, where StationNumber is a text or numeric parameter of that page.

Format(width, precision, value)	Turns a numeric value into a text string, having the specified width and precision (number of decimal points). `Format(5, 2, [TankLevel_1])`

	Returns (for example): "30.36"
Replace(sentence, start, length, find, replace)	Searches the sentence, starting at the Start character and continuing for Length characters, looking for every instance of Find and replacing it with Replace. Note that character counting begins with 0. `Replace("This is good", 1, 12, "is", "was")` Returns: "Thwas was good" (Note that every instance of "is" is replaced. This may have unintended consequences.)
SubStr(sentence, start, length)	Returns a substring of Sentence, beginning with the Start character and running for Length characters. `Substr("on a Halifax pier", 5, 7)` Returns: "Halifax"

Related Functions:

...String And Buffer

Time and Date in Expressions

VTScada counts time in seconds (and fractions of a second). This is useful for calculating how long a pump has been running, but is not useful for humans to view. Fortunately, VTScada also provides functions that will translate the raw time into a format that is human-friendly.

Now(interval)	Returns the number of seconds elapsed so far today, updated every (interval) seconds.

Time(seconds, format)	Return the number of seconds since midnight in a format that's easier for a human to read and understand.
	Examples:
	`Time( Now(1), 2)`
	Returns: 21:35:00
	`Time( Now(1), 7)`
	Returns: 09:35 PM

Today()	Returns the date as a count of days since January 1, 1970.

Date(Daycount, format)	Turns the result of the Today() function into a format that is easier for a human to read.
	`Date(Today(), 2)`
	Returns: 12/25/09
	`Date(Today(), 21)`
	Returns: December 25

A partial list of the formatting codes for the Time function is as follows. For a complete list of formatting codes and for other time-related functions, see Time and Date Functions. (Many time and date functions work only in Scripts, and thus may not be used in an expression.)

Format	Example	Code to use
	No Time	0
hhmmss	173500	1
hh:mm:ss	17:35:00	2
hr:mm:ss HH	9:35:00 PM	6

The following table provides a partial list of formatting codes for the Date function.

Value	Date Format	Value	Date Format
0	No date	21	mmm..m d
1	yymmdd	22	mmm yyyy
2	mm/dd/yy	23	mmm..m yyyy
3	mm-dd-yy	24	dd/mm
4	mmm d, yyyy	25	dd-mm

Related Functions:

...Time And Date

Examples of Expressions

The following examples may be adapted to your application, or they may spark an idea for what you can achieve using expressions.
Simple math. Add two AI tag values together:

```
[Valve1Flow] + [Valve2Flow]
```

Using logic to check conditions. All the tags in this example are Digital Inputs that will be 1 (TRUE) or 0 (FALSE). In this test, we're checking whether either both of Valves 1 and 2 are open, or else if the overflow valve is open.

```
( [Valve1Status] && [Valve2Status] ) || [OverflowValveStatus]
```

Limit a return value to be no less than 0

```
Max([Valve1Flow], 0)
```

Write a 1 when a test is true, otherwise write nothing. The following example takes advantage of the fact that VTScada will not write an Invalid.
If the following were being used to control a pump for example, it would turn the pump on when the flow rate went above 1500, but would not turn the pump off.

```
[Flow] > 1500 ? 1 : INVALID
```

Conditional math. Add together only the MVAR values (in AI tags) for generators whose breakers are closed (DI tags == 1)

```
([Gen1Breaker] == 1) * [Gen1Mvar] +
([Gen2Breaker] == 1) * [Gen2Mvar] +
([Gen3Breaker] == 1) * [Gen3Mvar]
```

Get the value of an INI file variable for display:

```
Scope(\Code, "MyAppVersion")
```

Build a text string based on values in bits of registers read from a PLC.

```
Bit([AI3], 15) ? "On " : "Off "
```

Calculate a steam flow from a DP cell reading on an AI tag:

```
SQRT((([FT-518-RAW])/4095)*100) * 1000
```

Calculate a steam flow from a DP cell reading on an AI tag. Force to 0 if a DI tag is not on:

```
[BLR-001]==1 ? SQRT([FT-506-RAW]) * 3000 : 0
```

Figure out the bit number that is turned on in a word read from a PLC into an AI tag:

```
Int((Log([AI3] % 65536) / Log(2)) + .5) + 1
```

Create a time string based on an AI tag, AI3, and replace the 00 hour reading with 24:

```
Concat("HE ",Replace(Substr(Time((Scope([AMA.STA.MW-TS]) - (9 *
3600)) % 86400, 2),0,2),0,2,"00","24"))
```

Get the name of the PC that the application is running on:

```
\Code\RPCManager\WkStnName
```

Get the version of VTScada that is running:

```
Version()
```

Create a flag that toggles every 10 seconds

```
Latch(AbsTime(1, 20, 0), AbsTime(1, 20, 10))
```

Check if any user is logged on to a client PC. This example returns a message, but you could just as easily return a 1 or 0 to determine whether a control action should proceed.

```
\Code\SecurityManager\IsLoggedOn() ? "" : "Authorized Access Only.
Please Log On."
```

Get the name of the operator who is logged in:

```
\Code\SecurityManager\GetUserName()
```

Given an application with the following set of application privileges, you want to configure the Push Button widget so that confirmation is required when the button is pressed by anyone whose account does not have the "Confirmation Not Reqd" privilege.

```
<SECURITYMANAGER-PRIVAPP>
PrivBitsTotal = 3
PrivDesc0 = Page Priv,0
PrivDesc1 = Operational Priv,1
PrivDesc2 = Confirmation Not Reqd,2
```

Confirmation Not Reqd is enumerated as privilege 2, therefore its index value is 16 + 2 = 18. The expression for the Confirmation Dialog parameter of the widget would be:

```
1 - PickValid(\SecurityManager\SecurityCheck(18, 1), 0)
```

In a parametrized page, find the name of the tag that was used for a given parameter:

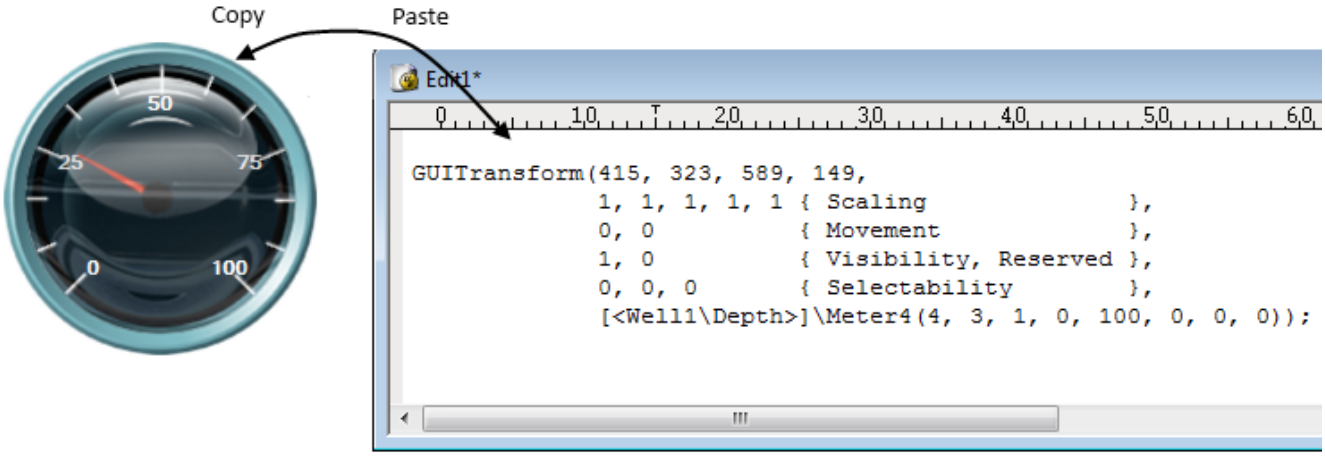
```
\NameOfGivenParameter\Name
```

In a parametrized page, where a parameter is of type tag, use the description of the tag as part of the title:

```
concat("Details for: ", \NameOfGivenParameter\Description)
```

The VTScada API

Script code lies behind everything that you see in VTScada. An interesting exercise to illustrate this is to copy any object from a VTScada application page (Ctrl-C) and paste it into a text editor (Ctrl-V). Rather than seeing the same image that was on the page appear in the editor, you will see the code that animates that image. The same will work in reverse.



Every VTScada page is a **module**[1], stored in a separate file on disk, and each of these files is a plain-text source file of VTS code. Examining the source code for a page (or a user-defined drawing object) is a good way to begin learning the language.

If you are familiar with programming other languages, you will quickly discover that VTScada works in a unique way. Other than blocks of **script**[2] code, everything runs in what is termed, "**steady state**[3]". This is a

---

[1]A collection of states, scripts, variables, parameters, comments and possibly other modules, all of which make up a VTS program. Modules are separate tasks that run simultaneously in an application. Behind every page is a module and behind every tag on that page is an instance of another module (usually with submodules controlling separate tasks).

[2]VTScada statements that are evaluated in order as written. Scripts are triggered by an IF statement within a VTS state.

[3]The operating condition of the code within a State.

form of event-driven programming. In a block of code (a state[1]), such as that used to display the objects in a page, each line will run once (for example, to initially display the page), and then not run again until a variable within the line changes value (for example, an I/O tag obtaining a new reading from the PLC). In steady state, each individual line of code (called a "statement") may execute at any time, and will do so independently of every other statement. This makes VTScada extremely efficient, but you will need to learn new coding techniques.

Study Guide

To learn the VTScada script language:

- Start with the fundamental components of expressions, states and steady state, and VTScada modules.
- Review the style recommendations for writing VTScada code.
- Learn how to add your code into an application.
- As you write your code, refer as necessary to the definitions of variable types, operators, and functions.

When you have mastered these basics, refer to the remaining chapters of this guide as needed for examples and guidance when creating custom tags, user-interface wizards, device drivers, etc.

Related Information:

...Parts of a VTScada Program

...States and Steady State

...Action Triggers and Scripts

...VTScada Modules

...Functions

... Threading

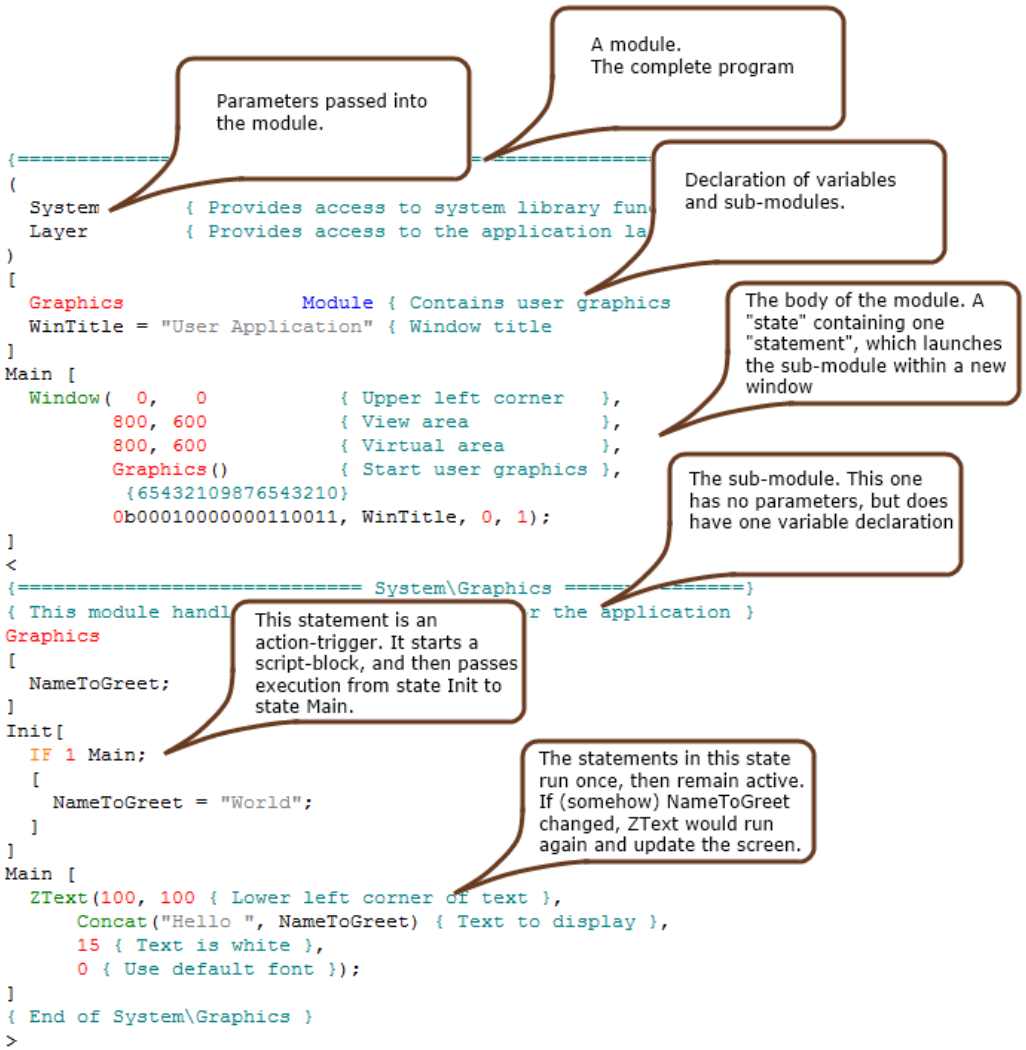
...Operators in Statements

---

[1] A collection of statements, grouped together within square brackets and given a name. A state is the part of the module that performs a task. Modules can have many states, but only one may be active at a time.

Parts of a VTScada Program

This example shows a complete VTS script program. When run, it will display the words "Hello World" within a small window. A description of the various parts follows the example.




```
{=============                                    ==============
(
  System        { Provides access to system library fun
  Layer         { Provides access to the application la
)
[
  Graphics                Module { Contains user graphics
  WinTitle = "User Application" { Window title
]
Main [
  Window(  0,   0           { Upper left corner   },
         800, 600           { View area           },
         800, 600           { Virtual area        },
         Graphics()         { Start user graphics },
          {65432109876543210}
         0b00010000000110011, WinTitle, 0, 1);
]
<
{============================== System\Graphics ======  =======}
{ This module handl                              r the application }
Graphics
[
  NameToGreet;
]
Init[
  IF 1 Main;
  [
    NameToGreet = "World";
  ]
]
Main [
  ZText(100, 100 { Lower left corner of text },
      Concat("Hello ", NameToGreet) { Text to display },
      15 { Text is white },
      0 { Use default font });
]
{ End of System\Graphics }
>
```

Building it up piece by piece...

- The smallest bit of code that will return a value is called an **expression**[1]. An expression might be simple math, a call to a **function**[2], or some other programming construct.
- A **statement**[3] is a complete line of code (possibly running over several lines in the editor). Statements are built from one or more expressions.
- Statements are always found within a named **state**[4]. Code within a state is executed in **steady state**[5], which means that after running once when the state is first activated, the statement lies dormant until triggered by a change to a variable within it.
- Statements that must run in a given order, or run a predictable number of times rather than being event-driven, are put into a **script**[6] block within a state. A statement called an *action-trigger* is used to start a script block running. Action-triggers are also used to transfer execution from one state to another within a module.
- The entire block of code that holds one or more states, as well as variable declarations, etc. is a **module**[7]. Within a module, only one state can be

[1]Any calculation that returns a result. Examples include a call to a function, assigning a value to a variable, and mathematic operations.
[2]A module that returns a value. For example, math functions such as Sqrt(), Min() and Abs().
[3]A command made of one or more expressions and function calls, always ending with a semi-colon, directing VTScada to perform an action.
[4]A collection of statements, grouped together within square brackets and given a name. A state is the part of the module that performs a task. Modules can have many states, but only one may be active at a time.
[5]The operating condition of the code within a State.
[6]VTScada statements that are evaluated in order as written. Scripts are triggered by an IF statement within a VTS state.
[7]A collection of states, scripts, variables, parameters, comments and possibly other modules, all of which make up a VTS program. Modules are separate tasks that run simultaneously in an application. Behind every page is a module and behind every tag on that page is an instance of another module (usually with submodules controlling separate tasks).

active at a time. (You can get around this limit by having a module launch a submodule with its own active state.)

States and Steady State

The executable code of a module is contained within one or more named states. Each state contains a collection of instructions (statements) describing what the module is to do while that state is active.

Note: Only one state in a module may be active at any time. While a given state is active, all other states in the module are ignored.

The first state found in a module is always the one that will run when the module starts. Execution can switch freely from one state to another based upon programmer-defined action triggers (See: Actions and Scripts). Until an action trigger occurs, the active state will remain active. When a state becomes active, each statement within it will be executed exactly once. The state remains active, but no statement is executed again until one of the variables within it changes. This is referred to as "steady state" in VTScada.
If two or more instances of a module are running, a different state may be active in each.
A state is defined using a formal structure as follows. Note the square brackets that enclose the body of the state.

```
StateName [
    Statement1;
    Statement2;
]
```

The following rules apply to states:

- There is no limit to the number of states in a module (other than the computer's RAM).
- Every state must have a State Naming Rules, and enclose all of its statements in square brackets.
- In any module, only one state may be active at a time.
- All statements within a state will execute once when the state first activates.

- Following the first run-through, any statement will execute again only when triggered by a changing variable value. In steady state, there is no way to predict which statements will be triggered in which order.
- If the module must do two things at once, create a submodule and call it from the state.
- In the module for a page, the z-order of graphics matches the order of their statements.
- Excepting the previous point, the order of the statements is largely irrelevant.
- If order of execution matters, use a script.

It is common for a module to have only one state. Modules in which the operations remain constant fall into this category. Typically, modules that display pages or that handle alarms, PID control or I/O need the same instructions to remain active regardless of plant activities or operator inputs. For example, it would be undesirable for a PID instruction to be placed in a module with several states since the PID control action would only occur when the module was in the state containing the PID instruction.

The mechanism that transfers control from one state to another is the same one that controls when a **script**[1] should run: an Action Trigger.

Related Information:

...State Naming Rules

...Event-Driven Execution and Efficiency

State Naming Rules

State definitions begin with a name. Each state in a module must have a unique name, however it is legal (although confusing) for a state to have the same name as the module.

- State names must be a single word.
- Use an underscore or CamelCase to indicate multiple words

---

[1] VTScada statements that are evaluated in order as written. Scripts are triggered by an IF statement within a VTS state.

- State names may not include most symbols, such as #&-+=<>{}[](), etc.
- State names are not case-sensitive. "Main" is the same as "main".
- State names may be numeric, so long as you do not include a negative sign.

In general, state names should be:

- Short
- Descriptive – of the state's purpose
- Verbs – when the state performs a task

You will often see modules that contain states named "INIT" and "MAIN". There is no functional significance to these names, but by custom, INIT is used for initialization tasks and MAIN is used for the state that forms the main body of the module.

Event-Driven Execution and Efficiency

In steady-state, statements do not follow a strict sequential execution after the initial run-through. All statements in a particular state are executed once, in order, when that state first becomes active. After the first execution, a statement or action trigger is executed again when the value of any variable in the statement changes.
It is important to watch for possible race conditions. For example, if a state called Monitor contains two action trigger statements:

```
If HighAlarm Shutdown;
If HighAlarm StartPump;
```

Both of these actions change to a new state on the same condition. It cannot be predicted which will execute first, and the final state will be either Shutdown or StartPump. This is called a "race condition", because the module depends on which action trigger wins the race. When designing a state, assume that all statements and action triggers will execute simultaneously. Although this is not strictly what happens, it is a good design strategy to use.
Changes to a variable's value propagate through the system. For example, consider the statement:

```
X = Y + 4;
```

Every time the variable Y changes, all statements that depend on the variable Y will be updated, including the assignment to X. Then, because X changed, all statements depending on the value of X will be updated.
This rule may affect how you write seemingly simple statements. For example, if the following were to appear in steady state code, X would continuously increment.

```
X = X + 1;
```

Presumably, you would only want X to increment in response to some specific condition or at a particular time. For this purpose, you could use a script block.
This event-driven model results in VTS being very efficient. For example, an active state could contain hundreds of output statements that display one variable each. If all statements were to execute, it would take a certain length of time. If a looping mechanism were used to continuously poll for which variables had changed, then much more time would be required. But, in VTScada, if only one variable changes, the CPU need only process one statement instead of hundreds. This means that the response time for the page is cut to one percent or less of the time it would take to update all statements..
Another example is a state with many actions, each looking at the same variable:

```
If Stage == 1 Alarms;
If Stage == 2 Mixer;
...
If Stage == 100 TankFarm;
```

In this example, every time the variable Stage changes, VTScada attempts to execute each of the action triggers once, in an unknown order. This does not necessarily mean that all 100 actions are checked -as soon as one is found to be true and it contains a state change, it will stop this state and start a new one. When the state is stopped, the remaining actions are no longer checked. It could be possible that all 100 actions would have to be checked, however, it is more likely that an average of 50 actions would be checked, based on a uniform random distribution of Stage.

When designing a module, it is a good idea to identify activities that take place continuously, such as a flashing lamp or PID loop, and keep them separate from activities that start on a trigger, such as starting a pump when a level is too high, or switching the graphics screen at the press of a button.

Action Triggers and Scripts

An action trigger is the instruction that passes control from one state to another. Being a trigger, it requires a conditional expression to signal when the state transfer is to occur.
When the purpose is to transfer control to another state, an action trigger always takes the form:

```
IF conditional_test NextStateName;
```

Note the semi-colon after the name of the state that control will be transferred to. This line of code is a statement and must follow the rules for all statements. Since it runs in steady state, it will be triggered whenever the variables in the conditional expression change and the test then becomes TRUE.
The same code can also be used as the signal to run a script. A script is a set of statements that will run in order for as long as the trigger condition is true. Scripts are used within states in order to execute code where the order of the statements matters, and where you need to control how many times the code will be repeated.
VTScada includes functions that are designed to run only in steady state and functions that are designed to run only in a script block. When reading about a function in the reference section of this guide, take care to note the Usage field.
To run a script, the action trigger is modified to include the script statements, within a set of closed parenthesis, and following the IF statement and its semi-colon. The general form is as follows:

```
IF conditional_test NextStateName;
[
 first_script_statement;
 second_script_statement;
 ...
]
```

A common example is an initialization state. It's purpose is to initialize variables, and perhaps open a file stream or other I/O. Those tasks must be done once and once only, therefore it makes sense to put them in a script rather than in steady-state code.

```
InitState { name of the first state to run}
[
 IF 1 MainState; { "IF 1" is guaranteed to be true, therefore the
script will run and control will then switch to MainState }
 [ { script that should be run once }
   x = 1;
 { other initialization tasks ... }
 ]
]
MainState { name of the state that does the work }
[
 { code that does the work }
]
```

Rules for Action Triggers

- When the next state name is provided, the script block is optional.
- When a script block is provided, the next state name is optional.
- When both are provided, the script block will be executed exactly once, and then control will be transferred to the next state.
- If there is a script block, but no state to transfer to, then you must ensure that the conditional test will become FALSE after one or more iterations through the script. Otherwise, it will run indefinitely.
- While a script is active, no other statements in the state will execute (excepting the IF action trigger).

Note: Note: If the action trigger includes a destination state, all code in the current state excepting the script block will STOP. This means that variables outside the block will immediately become invalid and modules that were called from that state will terminate. If code in the script block depends on any of these variables, use a ForceState function

---

within the script-block instead of a destination state in the action trigger.

A typical action might look like the following example. Here, the first line contains both the action trigger "If TimeOut(1, 5)" and the destination state "Start". The square brackets delineate the script block.

```
IF TimeOut(1, 5) Start;
[
  X = 0;
]
```

The trigger statement will become true five seconds after the state containing this code becomes active. When this happens, X will be set to 0 and execution switch to a state named Start.

Related Information:

...The Trigger

...The Script Block

The Trigger

```
IF Trigger DestinationState;
[
   { script block }
]
```

The trigger is any logical expression that determines if a state transfer is to take place. Nothing happens as long as the trigger expression evaluates to a logical false (0).
While an action trigger is false, no action is taken. When it becomes true, the following occurs:

1. If there is a destination state, the active state (and all its statements) is stopped(*).
2. The script block (if any) is executed in order from the top of the list to the bottom.
3. The destination state (if any) is started as the new active state.
4. If there is no destination state, the script will be executed repeatedly while the action trigger remains true.

This will continue until the action trigger becomes false. Take care not to inadvertently create such a repetitive loop since it will consume processing time and greatly degrade the overall system performance.

(*)Note: If the action trigger includes a destination state, all code in the current state excepting the script block will STOP. This means that variables outside the block will immediately become invalid and modules that were called from that state will terminate. If code in the script block depends on any of these variables, use a ForceState function within the script-block instead of a destination state in the action trigger.

Note: When combining function calls and other operations in an action trigger, use care to follow the rules of operator precedence to avoid unexpected results.

"1" is a common trigger condition to use when you want to force a script to execute, *followed by a state-change*. Use care when this is done inside a module that is run as a subroutine. It is essential in this case, that the subroutine's Return statement be located in code that will not remain active. This can be done by ensuring that the return statement is in a script and that the script does not remain active. Consider the following four examples of subroutine modules:

```
ExampleSubroutine1
(
    Input;
    A;
)
[
    Result;
]
 StateOnly [
    IF 1;
    [
       Result = SomeFunction
(Input, A);
       Return(Result)
    ]
]
```

This is dangerous. If this subroutine is called from a script, all is well. But, if this is called in steady

```
ExampleSubroutine2
(
    Input;
    A;
)
[
    Result;
]
 StateOnly [
    IF Watch(1);
    [
       Result = SomeFunction
(Input, A);
       Return(Result)
    ]
]
```

In this version of the module, the script will be executed only once. The IF statement resets the value

state (for example, in a Calculation tag), then the script within it will execute continuously, as fast as VTScada can go, resulting in very high CPU usage.

```
ExampleSubroutine3
(
    Input;
    A;
)
[
    Result;
]
 StateStart [
    IF 1 StateDone;
    [
       Result = SomeFunction
(Input, A);
       Return(Result)
    ]
]

 StateDone [
 { empty state }
]
```

In this module, the script will also be executed only once, but an extra state is required. While this works, it is generally regarded as poor practice.

of the Watch() function to Invalid after it runs once.

```
ExampleSubroutine4
(
    Input;
    A;
)
[
    Result;
]
 StateOnly [
    IF Watch(1, Input, A);
    [
       Result = SomeFunction
(Input, A);
       Return(Result)
    ]
]
```

In this module, the script will execute when the module is first started, and (if called from steady-state) will then re-execute whenever either of the inputs change.

This is expected behavior that you can use to your advantage. Latching and Resetting Functions

The Script Block

A script block contains a set of instructions that are executed in order when the block is activated. The block is activated when its action trig-ger's condition becomes true.
Standard programming constructs such as If-Else conditions and Do-While loops may be found in a script block since the order of execution

is predictable. Script blocks are also used for most file I/O, for the same reason.
Any number of statements may be present in a script, including none. The only limitation upon the number of statements in a script is the available memory (RAM).
If the action trigger includes a transfer to another state, then the script-block will execute exactly once before the state transfer occurs.
If the action-trigger does not include a transfer to another state, then when the script block reaches its final statement, if the action-trigger's condition remains true, the script-block will run again. Take care to not create a script that run continuously.
While a script block is active, no instructions outside of the block will execute. If the action trigger includes a destination state, then statements outside the block will stop and all variables will go to INVALID. For example...

```
X = DBSystem(...);
If A SomeOtherState;
[
   // X is now invalid
]
If B;
[
   // X would still be valid (if it was valid)
   ForceState("SomeOtherState");
]
```

If either A or B becomes true, a script block will execute and then control will transfer to the destination, SomeOtherState, but the action is not quite the same. When A becomes true, X immediately becomes invalid and is not available to code in the script block. When B becomes true, X will not update (that statement being outside the script block) but it will also not become invalid until after the call to ForceState.
If B does not force a state change, and does include code that will change B back to Invalid thus stopping the script-block, then the current state will resume and X will still have its value.
There is one exception to the rule about scripts being executed in their entirety: If the module that contains the script is a "launched module"

and the script contains a Slay statement, then the module will be stopped immediately and the script execution along with it.

VTScada Modules

A VTScada module is the collection of all of components defined in the preceding topics (state[1], script[2], statement[3], expression[4]) into a program that does something. A formal definition is: a state-logic control program.

For example, every page in your application is a module. Each user-defined widget is also a module, as is every tag. An application is built with many program modules, each providing the instructions for a separate task to be done by the system.

There are no limits to the types and numbers of modules that you can define in an application. It is also common to find many separate instances of the same module running simultaneously in an application - for example, every Analog Status tag that you define is a separate and independent instance of the Analog Status module.

Modules are very often built using several submodules. Since only one state may be active in a module at any given time, submodules are required in order for the module to do two things at once. For example,

---

[1]A collection of statements, grouped together within square brackets and given a name. A state is the part of the module that performs a task. Modules can have many states, but only one may be active at a time.

[2]VTScada statements that are evaluated in order as written. Scripts are triggered by an IF statement within a VTS state.

[3]A command made of one or more expressions and function calls, always ending with a semi-colon, directing VTScada to perform an action.

[4]Any calculation that returns a result. Examples include a call to a function, assigning a value to a variable, and mathematic operations.

an existing instance of an Analog Status tag will continue to read values from the I/O driver while its configuration dialog is open. Within the Analog Status module, there is one module that handles I/O and another module is used to display the configuration dialog.

Module Files:

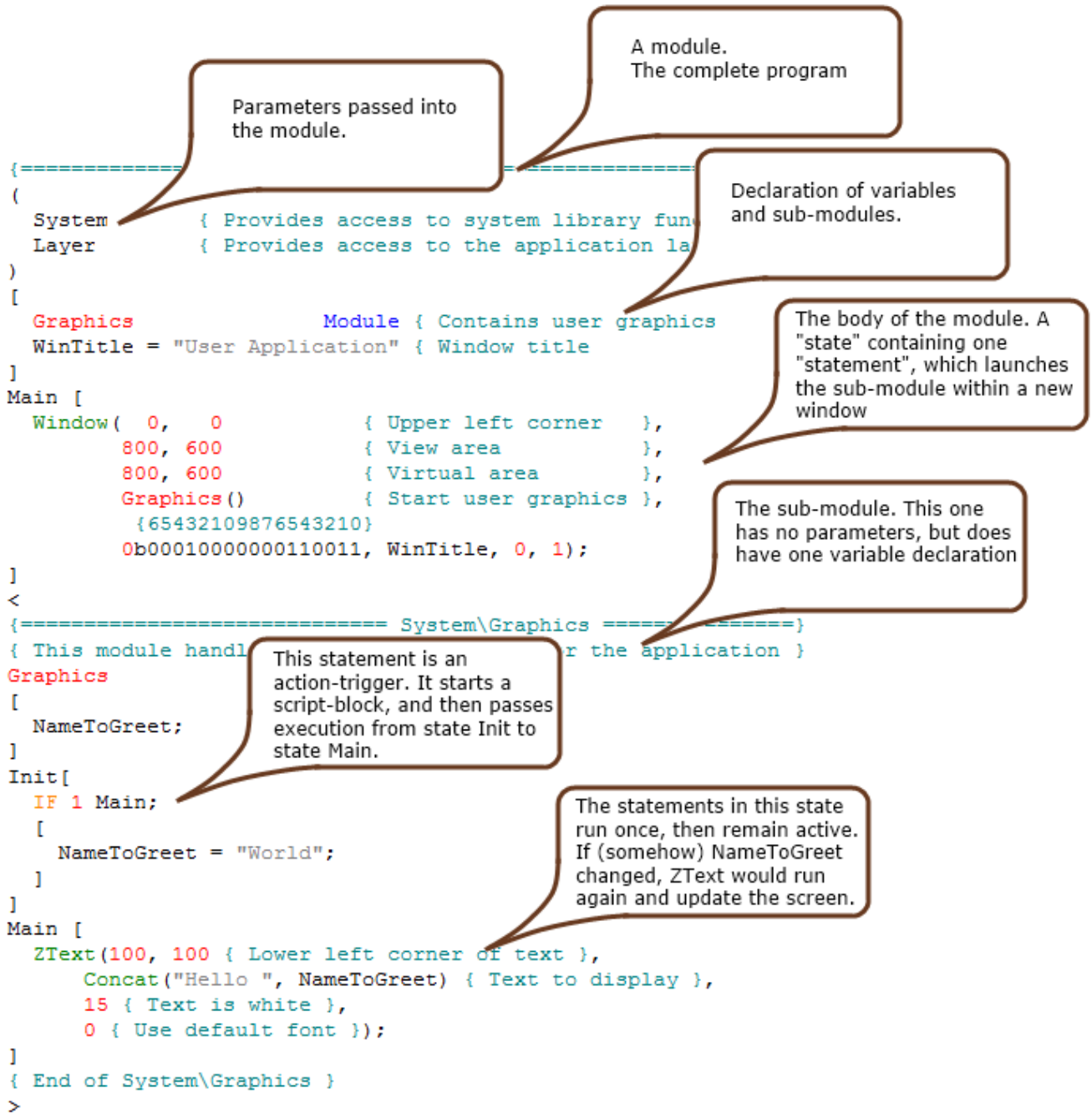
Source code for modules is stored in plain-text .SRC files. Compilation (part of the "Import File Changes" process) produces a set of files known as .RUN files. Source files can be excluded from the File Manifest before building a ChangeSet of an application for distribution. Only the .RUN files are required in order for VTScada to run the module.

Module Calls:

How a module is started is almost as important as the code within the module. Very different behaviors can be obtained depending on whether the module is called from script or steady state.

Module Structure:

There is a formal definition for how the parts of a module are put together.




```
{============
(
  System          { Provides access to system library fun
  Layer           { Provides access to the application la
)
[
  Graphics                Module { Contains user graphics
  WinTitle = "User Application" { Window title
]
Main [
  Window(  0,   0           { Upper left corner   },
         800, 600           { View area           },
         800, 600           { Virtual area        },
         Graphics()         { Start user graphics },
          {65432109876543210}
         0b00010000000110011, WinTitle, 0, 1);
]
<
{============================ System\Graphics ======  =======}
{ This module handl                    r the application }
Graphics
[
  NameToGreet;
]
Init[
  IF 1 Main;
  [
    NameToGreet = "World";
  ]
]
Main [
  ZText(100, 100 { Lower left corner of text },
      Concat("Hello ", NameToGreet) { Text to display },
      15 { Text is white },
      0 { Use default font });
]
{ End of System\Graphics }
>
```

In order from top to bottom, the parts are:

- A comment section, describing the module.
  While this has no effect on the module's operation, it is a vital component from the point of view of good coding practice.
- Reference box numbers, enclosed in parentheses. [Optional]
  Found only in modules that are used to draw graphics on a page. These four integer numbers within parenthesis, override the reference box that would

otherwise be defined by the total size of the graphic statements within the modules.

- Parameters section, enclosed in parentheses. [Optional]
  Not all modules will require a set of parameters. If the module does not have parameters, the parentheses are optional, and are normally not included.
- Variable section, enclosed in square brackets.
  All variables, constants and submodules used in the module must be declared.
  Constants must have their values assigned as part of the declaration. Variables may optionally be assigned values when they are declared.
  Sub-modules (if any) must declared and include the keyword "Module". Additionally, if the submodule is stored in another file, that file name must be provided.
- States.
  Each state in the module begins with the name of the state, followed by square brackets that enclose the code of the state. The first state found in the module will always be the first state to run. It is commonly called "Init", but the name has no special meaning.
- Statements. [Optional]
  Complete lines of code (possibly running over several lines in the editor) and ending with a semi-colon. Statements are built from one or more expressions.
- Scripts, enclosed in square brackets and following an action-trigger statement. [Optional]
  States may include script blocks if there is code that must be executed in a pre-determined order in response to a condition becoming true.
- Sub-modules, enclosed in angle brackets. [Optional]
  These are also referred to as "child modules". The first module within any file need not be enclosed in angle brackets, but each child module within the same file must be. Within each child module, the structure is the same as listed in the preceding points.

Related Information:

...Store and Declare Modules

...Types of Module

...Module Scope

...Constructors

...Destructors

...Reference Boxes in Graphic Modules

Store and Declare Modules

Where you will store your module code depends on the purpose of the module.
Pages are stored in the application's Pages folder and you must register them with the application by importing them into the Idea Studio. (This serves to declare the module in the (PAGES) group of the AppRoot.SRC file.
User-defined widgets are stored in the root folder of the application, and must similarly be registered using the Standard Library - User Draw Methods folder.
If you are creating a script application from scratch, you will probably add your code into the AppRoot.SRC file of the application. Or, you may choose to declare it within the Variables section of AppRoot.SRC and store the actual code in one or more separate files.
If your goal is to add functionality to an application, then you must save your module code in its own file(s). You will then declare the code within the appropriate section of the application's AppRoot.SRC file. For example, new tag modules are declared in the (POINTS) group and new report modules are declared in the (PLUGINS) group.

Types of Module

VTS defines module types according to how they are called and their behavior when called.

Called Module

Called modules are started from steady state. A set of parenthesis must always follow the module call, regardless of whether parameters are

included within in them.
A sample module call:

```
Motor(398, 765 { X-Y coordinates },
      13 { Color },
      MtrStat { Motor status },
      MtrAmps { Motor current });
```

The state containing the called module call is the "calling state". Called modules may themselves contain module calls, but take care not to call the first module from within the second. A circular or a recursive situation will result in a module calling itself repeatedly until a stack overflow fault occurs and the application crashes.
When the state containing a module call activates, an instance of the called module will start. When the state containing the module call stops, the called module's instance will also stop. Thus, called modules are active only while the calling state is active.
Parameter values are passed by reference. Any change to the parameter values in the calling or the called module will affect those in the other.

Launched Module

There are situations where you might want a module to execute whenever the calling module is active regardless of which state it is in. You may also have a situation where you want to create multiple instances of a module, where the number of instances to be started is not known until the application is running. Launched modules answer these needs.
A module is launched by placing its module call in a script, or by using the Launch function (which enables better control of the module instance's parent and caller). When a module is launched, the parameters of the module are evaluated at the moment in which the statement is executed and the values are set in the corresponding parameter variables in the launched module. (Pass by value) A later change in the calling parameters does not affect the parameters in the module. The return value for a launched module is always the object value of the launched instance.

The launched module is stopped either when the calling module stops (provided it is not a subroutine) or when a Slay statement is executed to explicitly stop the module instance. In the case of modules launched by a subroutine, the subroutine itself is not considered to be the parent or caller, but rather, the module will be launched with the caller of the nearest non-subroutine caller. This means that when the subroutine ends, any modules launched by it will continue running until its non-subroutine caller stops.

Since repeatedly executing the same module call in a script will create new instances of the launched module, any number of module instances can be launched without having to have a separate line of code for each instance.

Note: Launched modules must not have a Return statement in them. If they do, they will be considered subroutines.

Use care in how you write a Launch statement. For example:

```
X = Launch(Scope(<variable>, "Y", TRUE)...
```

If it is unknown whether Y exists in the module pointed to by <variable> or if in fact <variable> is valid, then setting the ScopeLocal parameter to TRUE as shown, may help to avoid undesirable results.

Subroutine Module

Subroutines are syntactically similar to modules that are launched implicitly by being called inside of a script. The difference is that subroutine modules have one or more Return statements in them. This causes their behavior to be different, since the calling module will suspend execution of the script that started the subroutine until the subroutine executes the Return. Once the Return is executed, the subroutine module is stopped immediately and the calling script resumes. This enables the building of modules that return a value that can be used for subsequent statements in a script.

A module that has a Return statement in it is only considered a subroutine, and will only behave in the manner as described above, if it is

called inside of a script. Program execution will not be suspended if the module containing the Return is called in steady state.
The other difference between subroutines and launched modules is how they launch other modules. A module launched by a subroutine will not take the subroutine as its parent or caller, but rather, will consider the nearest non-subroutine caller as its own caller.

Note: WARNING: Great care must be taken in the use of subroutines, since no other statements, I/O or alarms will be executed while a sub-routine is running. This means that if a sub-routine launches a module with the intention of waiting for the results of its execution, the application will hang, since even child modules will be blocked by their calling sub-routine. It is vital to ensure that a subroutine executes a Return statement since all other modules will be suspended until this occurs.

Queued Module

To understand queued modules, it is necessary to recall that any number of module calls can be created as statements in steady states. Normally, the number of concurrent copies (instances) of a module that can be created by module calls is limited by available RAM memory. Every module call starts its own instance when it is active. Prefixing a module definition with the keyword "Queued" makes that module a queued module and only a single instance of the module is allowed to run - all other module calls will return invalid values and do nothing. All module calls to that queued module that do not run will enter a queue for that module, waiting for a chance to run. Each successive queued module may run when the instance queued before it, stops.
Typically, the queued module will return a value to signal its calling module that it has completed its work. The calling module will change to another state, which will stop that instance of the module and make a space available for the next queued module call to run an instance. The next module to run will be the module which has been waiting the longest in the queue (i.e. the one that entered the queue first or earliest).

Every instance of a queued module contains its own variables, and behaves in every way like other modules, separate from other instances of the same queued module.
Note that queuing will occur only with modules called from a steady state; queued modules will not work on modules that are implicitly or explicitly launched.
Programmers familiar with much earlier versions of VTScada might look for "Fixed modules". "Fixed" is an obsolete term for what is now a queued module.

Threaded Module

Threaded modules are very similar to launched modules, and within the confines of their own thread behave much the same way. They may only be destroyed by doing a Slay from within their own code, if an external source has a copy of their object value upon which it may execute a Slay, or if their caller stops.
Threading has the advantage that no one module may completely monopolize the processor, even if it has accidentally been created with an if 1 condition. You will notice this if you write an application with an infinite loop in it and then attempt to debug the problem via the Debugger (see "Debugging and Analysis: Debugger). The debugger will function normally because it is in its own thread, and therefore is not blocked. Any other (non-threaded) statements in your application, however, such as buttons that perform certain tasks, will be entirely crowded out from getting access to the processor and will appear to be "locked up" or "frozen". Different applications are likewise executed in their own individual thread, so no one application will be able to block another from executing.
Why then shouldn't you thread all of your modules? The answer is twofold - firstly, threaded modules will have no predictability as to when they perform certain tasks relative to other statements being executed in the system. The second and most important reason why threads should be used sparingly is the overhead that each thread uses in terms of processor time as well as its RAM requirements. Apart from the time that it

takes to create and destroy a thread, there is also the time it takes for each switch between thread, as well as the time slice allotted to every thread for execution. Although a thread will surrender its allocated time slot with the processor if it has no tasks to perform, this in itself will have used up a certain amount of time. This means that the more threads that have been created, the more the application as a whole will be slowed down, as the processor keeps cycling through all threads, giving each equal opportunity to execute, no matter how unimportant its tasks may be relative to others in the application. Thus threaded modules performing trivial tasks could be taking away time from critical modules that may otherwise receive a larger portion of the processor's time. As a general rule, the number of threads created by an application should be fewer than six. They should strictly be reserved for crucial modules whose execution would otherwise block the application, or whose exclusion from executing would be detrimental to the proper functioning of the application.
It should be noted that VTScada provides utilities to assist you in troubleshooting threading in your applications. The Thread List utility supplies a list of the separate threads of execution for which VTScada is responsible within a local application (see Thread List Application).

Related Information:

...Declaring and Passing Parameters - The primary mechanism for handing information into modules.

...Parameter Metadata - Assign extra information when declaring parameters.

...Functions - A function is a module that returns a value.

Declaring and Passing Parameters

Modules often need to have information provided to them in order to function. Parameters provide the primary mechanism for passing information into (and sometimes out of) modules.

A parameter is declared by adding it to that module's parameter list, which is a list of variable names, separated from each other by a comma or a semi-colon, all of which are enclosed in a set of parenthesis and located at the beginning of the module.
The parameters declared within a module are referred to as the "formal" or "declared" parameters.
The parameters used when calling an instance of the module are referred to as the "actual" parameters.
The order of the parameters is significant since the value of the first actual parameter will be used for the first formal parameter, and so on. There is however, no requirement that the lists contain the same number of parameters.
If there are more formal parameters in the module than actual parameters in the module call, the extra formal parameters will have invalid values. If there are more actual parameters in the module call than formal parameters in the module definition, the extras will be ignored (although a module can access these undeclared parameters using the Parameter function).
Parameters act as placeholders for variables. The value within a variable that is passed to a module can be used and altered by the module as if it were a member variable of the module. If a constant, function, or expression is passed as a parameter, then that formal parameter's value cannot be altered.
In the case of parametrized modules, when the value of a variable passed as a parameter is changed outside the module, its value is updated inside the module. It is possible to reset the parameters to their original values using the ResetParm function, but most applications do not need this feature.
Parameters may be assigned default values in the module declaration. For example:

```
Motor(398, 765 { X-Y coordinates },
      13 { Color },
      MtrStat = 1 { Motor status },
      MtrAmps = 30 { Motor current });
```

If the actual parameters for MtrStat and MtrAmpts do not have values, or if these two actual parameters are missing when the module is called, then these will be given the default values of 1 and 30, respectively.

Parameter Metadata

You can assign metadata to module parameters when declaring them. This technique is used to ensure that extra information about the parameter is assigned when the module opens and before any of the state code runs.
Parameter metadata is declared using the metadata assignment operators: <: :>
For example, given the module X, with parameter A, a metadata value could be assigned as follows:

```
X
(
  A  <: 5 :>;
)
```

The function of the assignment operators is similar to that of the SetVarMetaData function. The values can be read using GetVarMetaData.
The most common use of parameter metadata is found in tag modules. The SQL data type of each parameter is assigned in the parameter declaration section:

```
(
  Name         <:TagField("SQL_VARCHAR(255)", "Name",         0 ):>
};
  Area         <:TagField("SQL_VARCHAR(255)", "Area",         1 ):>
};
  Description  <:TagField("SQL_VARCHAR(255)", "Description",  2 ):>
};
)
```

The SQL database conversion data for each parameter is recorded by instantiating TagField structures and assigning them in the parameter declaration section
See also: MetaData, SetVarMetaData and GetVarMetaData.

Module Scope

"Scope" refers to the ability of a variable or named module to be seen by calling code. For example, two modules may both declare a variable named "X". X will have a unique value in each module and will refer to a different memory address. Each version of X is *local* to the scope of the module that it is declared within.

A module may declare and use submodules. Those submodules (child modules) will also be able to see and use the variables of the main module (parent module). The parent is within the scope of the child. Variables declared in the child modules cannot be used by the parent unless directly referenced with the backslash scope resolution operator "\" (Child\Variable).

If one module needs to access variables or functions of another module, it is possible to do this by fully describing the scope of the variable or function being called. For example, many of the functions in the Function Reference will state the module they are a member of and provide an example of how they can be called: \AlarmManager\Acknowledge (AlarmName, EventTime, Operator);.

The following terms are used when describing scope:

Child

A submodule is a child of the module whose source file it is part of. Another term for a child module is a "member module".

Parent

A parent is a module that contains submodules.

Descendant

Sub-modules may themselves contain their own submodules. A descendant refers to any submodule of a parent.

Ancestor

Like "descendant", but looking in the other direction. Starting with any submodule, an ancestor is any parent module up the declaration chain. The root module is an ancestor of every module and every module is a descendant of the root module.

Member

Any named object – a variable, constant, module, etc.

Related Information:

...Scope Resolution Operators

...Module Inheritance

Related Functions:

... Scope

...LocalScope

Scope Resolution Operators

There are two scope resolution operators: the dot (.) and the backslash (\). The dot operator was added with VTScada version 11.2 and should be used when the intent is to reference a value within the reference scope. If there is no variable with a matching name in the current scope, the back-slash operator will find a variable with the matching name in a higher scope, whereas the dot operator will return invalid.
The dot operator is the equivalent of Scope(, , TRUE). For example, Obj.Value is equivalent of Scope(Obj, "Value", TRUE).
The scope resolution operator, a backslash (\ ), allows access to vari-ables and modules outside the current scope. This scope resolution oper-ator must be used with an object value, and a member name (either a variable or module name).
The backslash scope resolution operator may also be used to accomplish a feature called "late binding". A variable is a named storage location where a value is stored. The process of "binding" is the means by which the name and the storage location in memory are associated. Late bind-ing (also referred to as "dynamic binding") links a variable or object at run time. Early binding refers to the process of assigning types to vari-ables and expressions at compilation time.
When a scope resolution operator is placed before a variable or module name, such as in:

```
\VarName
```

it is considered the equivalent of writing:

```
Self\VarName
```

but it uses less RAM than the second statement. Early binding is typically more efficient than late binding as it reduces the amount of time required to set or retrieve a value, whereas late binding consumes more memory, and is slower than a direct variable reference; however, in some instances it is useful to reference modules and variables that are not in scope at compile time.

Related Functions:

... Scope

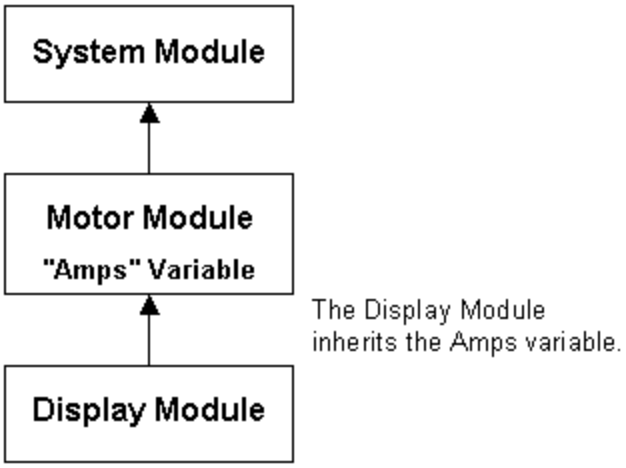
...LocalScope

Module Inheritance

When a member (a variable, constant or module) appears in a statement, VTScada looks first in the current module where the statement appears. If the member is found, it is used. If the member isn't found, VTScada looks in the module's ancestors, starting with its parent, until the member is found. If the member isn't found in the system module, VTScada reports an error: no such member exists. An ancestor's members are within the scope of all of its descendants.

Example:

A module called Motor is created as a member module of System. One of its member variables is Amps. Within Motor a member module called Display is created. Display has no member variable Amps. If a statement were entered in Display, which used the variable Amps, it will use its parent's variable Amps (which is Motor's member variable Amps).



When searching for a member, VTScada always searches ancestors, not descendants. So, in the example, if Display had the member Amps, and Motor had no member Amps, and a statement was entered in Motor which used Amps, VTScada would not find the Amps in Motor's child module Display.

If both Display and Motor had members called Amps, each would use its own member. This is because the current module is searched first. Note also that each Amps could have different values, and different types.

To use a descendant's member, a scope resolution operator, the backslash character '\', and an object value are needed, as described in the following topic.

Constructors

Constructors are subroutines that, if present in a module, are called automatically when the module is created. Constructors are useful for initializing variables, opening files, streams and other I/O connections, launching submodules or for registering the module with external services before the module is fully ready for operation.

Constructors differ from initialization states (the first state to run when a module is created) in that they will occur sooner. The constructor subroutine runs immediately as part of the launch, while there may be a delay before the first state in the module runs. Since constructors were introduced as part of VTS 10.1, you may see the use of a "Ready" flag in older code. This was used as a work-around to ensure that the first state

in the module had a chance to run before other code that depended on the completion of initialization tasks would begin.
Example:
Where MyModule is a launched module:

```
<
MyModule
(
 InstanceName;
 )
 [
  Constructor Module;
  CapsName;
]
Main [
  ...
]

<
Constructor
Main [
  If 1;
  [
    CapsName = ToUpper(InstanceName);
    Return(Invalid);
  ]
]
{ End of MyModule\Constructor }
>
{ End of MyModule }
>
```

The script code that launches the module might look like:

```
    MyObj = MyModule("MyInstance");
    InitCapsName = MyObj\CapsName;
```

In the above example, InitCapsName will be "MYINSTANCE", as the constructor subroutine is executed as part of the line of script that creates the object and assigns it to MyObj. Remember that subroutines take control of executing code until they are finished, thus guaranteeing that CapsName is set by the time the statement InitCapsName = MyObj\CapsName runs.
Constructors execute inside a CriticalSection, so it is impossible for external code to interact with a partially constructed module.
The parent object and caller object of a Constructor is the module being constructed.

Destructors

Destructors are subroutines that, if present in a module, are called automatically just before a module is slain or otherwise stopped. These ensure that cleanup tasks such as closing files, freeing memory and deregistering a module instance from external services are done even if the module is interrupted unexpectedly. (For example, a user closes a dialog box rather than finishing its task.)

```
<
MyModule
 (
  UniqueName;
)

 [
  Constructor Module;
   Destructor Module;
]
Main [
  ...
]
 <
Constructor
Main [
  If 1;
  [
    GlobalDictionary[UniqueName] = Caller(Self());
    Return(Invalid);
  ]
]
{ End of MyModule\Constructor }
>

<
Destructor
Main [
  If 1;
  [ { UniqueName had been added to a global dictionary as part of
MyModule.}
    DictionaryRemove(GlobalDictionary, UniqueName);
    Return(Invalid);
  ]
]
 { End of MyModule\Destructor }
>
{ End of MyModule }
>
```

In the above example, GlobalDictionary is a dictionary of all instances of MyModule, keyed by each module instance's UniqueName. The

Constructor makes sure that each instance of MyModule is present in that dictionary, and the Destructor removes the instance from the dictionary when the instance is slain.
Destructors execute inside a CriticalSection, so it is impossible for external code to interact with a partially destructed module.
The parent object and caller object of a Destructor is the module being destroyed.

Reference Boxes in Graphic Modules

If the purpose of a module is to display a graphic object, it is useful to define a reference box. This defines the rectangle to be occupied by all graphics drawn by the module, possibly including a margin around those objects. If a reference box is not defined, then VTS will automatically calculate one based on the graphics being displayed. Leaving the reference box to be calculated automatically can have a negative effect if the module contains various states with different graphic statements. If VTS must re-calculate the reference box as the module changes from state to state, transformations applied to the display can be affected.
Commonly, graphic objects are displayed using a GUITransform that sets the size, location, and other attributes. The module's reference box will be mapped to the bounding box defined and transformed by the GUITransform. So for example, if you had a module named PumpSymbol, you might display it as follows:

```
GUITransform(0, 150, 100, 50 { Bounding box of object },
             1, 1, 1, 1, 1 { No scaling },
             0, 0 { No trajectory or rotation },
             1, 0 { Object is visible; reserved },
             0, 0, 0 { Graphic cannot be focused },
             flow = PumpSymbol(1, amps) { A sample module call });
```

When a GUITransform is applied, the module will be scaled such that its reference box will exactly fill the reference box of the transform, and that will be acted upon by the other transform parameters. There is a clear advantage to having the graphic object's reference box remain constant, regardless of the active state.

The reference box is defined using constants (numbers or defined constants) that are enclosed in parentheses and placed immediately after the module's name in its definition. The x and y coordinates of the reference box corners are defined in the order LeftReference, BottomReference, RightReference, TopReference. Variables and expressions may not be used.
For example:

```
PumpSymbol
(0, 100, 100, 0) { reference box }
( { parameters }
  State { current pump state  },
  Amps  { Amperage to display }
)
[ ...
```

See also, SetModuleRefBox, but this is rarely used.

Related Information:

...Reference Boxes for Graphics Modules

...Use Scaling to Position Graphic Objects

Functions

A function is a named operation that may return a value, perform an operation or both. For example, the square root of a number is returned by a function named Sqrt. The Beep function will cause a tone to sound. Function names are not case-sensitive in VTScada.
Some examples of functions are:

```
Sqrt(10);
Log(X);
Limit(X, 0, 100);
YLoc();
YLoc;
```

Note the use of commas to separate parameters when more than one is required. If a function does not require parameters, you may omit the parenthesis without affecting operation, but this is discouraged as a matter of practice.

Functions may be used as parameters for other functions. These functions may then be used in other expressions, etc. There is no limit on the level to which functions and operators may be nested and combined, however you should strive for clarity by limiting the level of nesting.
You may define your own functions by creating subroutine modules

Related Information:

...Types of Module - A function is a module that returns a value.

...Function Parameters - Declaring pass-by-reference and pass-by-value parameters.

...Latching and Resetting Functions - Some functions will stay set, and some will reset after being called.

...Considerations for Graphics Functions - Preparation for and proper use of.

Format Examples for Functions

The format example, provided for every function, also provides relevant information about how to use the function and the library that the function is a part of. The indication of the library is especially important to anyone writing a Script-layer based application.

Optional Parameters

For most function examples, some of the parameters will be shown inside square brackets. These parameters are optional. If the default values, as described in the parameter descriptions, will serve for your purpose, then you may leave the parameters out. If you want to specify some of the optional parameters, then you must provide all the parameters between the last one required and the optional parameter you want to specify. Use Invalid for each of the intervening optional parameters that you do not want to specify.
Examples:

```
\System\DropList(X1, Y1, X2, Y2, Data, Title, Index, FocusID, Trig-
ger, NoEdit, Init, Variable [, DrawBevel, VertAlign, AlignTitle,
Style, BGColor, FGColor]);
```

All the parameters from DrawBevel onward are optional and may be left out of the function call. Assuming that valid values have been defined for the required parameters, this function will work if used as follows:

```
\System\DropList(X1, Y1, X2, Y2, Data, Title, Index, FocusID, Trig-
ger, NoEdit, Init, Variable);
```

If you wanted to draw a drop list with an orange background, and did not care about any of the other parameters, you could use:

```
\System\DropList(X1, Y1, X2, Y2, Data, Title, Index, FocusID, Trig-
ger, NoEdit, Init, Variable, Invalid, Invalid, Invalid, Invalid, 135)
```

Will this Function Work in a Script Application?

Most development work is done within standard applications (those based on the VTScada layer), and the documentation is written from that point of view. Script applications will not have access to function libraries that were created explicitly for the VTScada layer. Do not assume that any function will work in your script application until you have tested it. It is possible to offer general guidelines for recognizing which functions are likely to work in a script application, but again, you should always test first. When testing the function in a script application, try first with the format as shown. If the test fails, try using the function with the prefix \Layer\.

- Functions that are part of the \System layer will work in script applications.
- Most, but not all, of the basic string handling, math, time and date functions will work in a script application.
- If the format example begins with a backslash (\) and is not part of the \System layer, then it is likely that the function will not work in a script application, but test to be sure.

Function Parameters

In most cases, a function will have a fixed number of parameters. The order in which the parameters are listed is significant since VTScada interprets the meaning of each parameter according to its position in the list.

Some functions have optional parameters that may be entered or omitted at the user's discretion. In the function listings, later in this guide, optional parameters are shown enclosed in square brackets ([ ]). If using a function that has several optional parameters, and you wish to specify only the first and last, use INVALID for the intervening parameters that you do not want to specify.

Pass-by-reference / pass-by-value

- When a function is called from steady-state, parameters are always passed by reference.
- When a function is called from script, parameters are always passed by value.

You can override this by using pointers for parameters in script code, thereby achieving a pass-by-reference. In steady-state, you can effectively pass-by-value by using a copy of a value rather than the original.

Note: for some data types, all assignments are made by reference, not by value. This includes Dictionaries and Arrays.

If a function has no required parameters, the empty parentheses following the name may be omitted while retaining exactly the same operation. It is however, good programming practice to always include the parenthesis as a matter of style.
Functions may be used as parameters for other functions. These functions may then be used in other expressions, etc. There is no limit on the level to which functions and operators may be nested and combined, however you should strive for clarity by limiting the level of nesting.
Examples of functions nested within functions :

```
Sin(Sqrt(X) * 5);
Limit(Cos(X / 180), 0, Log(Sin(X / 180) + 2) + 1);
```

Latching and Resetting Functions

Most functions, called in steady-state, will change in value as their parameters change. For example, Y = SQRT(X). As X changes in value, so does Y.

All functions and statements are reset when the state that contains them is entered. They may also contain an enable parameter that enables them to be reset. For example, the AbsTime() function will not start counting time until its Enable parameter becomes TRUE.

VTS includes a class of functions referred to as the automatically-resetting functions. These are designed such that their value will latch once set, and will not change with future parameter changes. AbsTime() again is an example - once the designated time has arrived and the function becomes true, it will remain true until reset.

The automatically-resetting functions will reset when used as the action trigger of an IF statement and the condition becomes true. For example, in the case of MatchKeys, an action trigger might test for the operator input of a specific single key. If the intent of such an action is to perform a one-shot event, such as increasing a set point by a fixed amount, the action trigger must be false the next time it is tested or else the script will be executed innumerable times for just a single keystroke. Since the MatchKeys function is automatically reset when the IF statement containing it becomes true, the action script will execute only once when the key is hit.

Note: Be aware that an IF statement will attempt to reset any subroutine of yours that is used as the condition for the action trigger. See: Action Triggers and Scripts

Some functions include the ability to reset themselves. For example Latch(). The second parameter to this function is a 'reset', which when true, resets the function for the next go-around.

Examples:

```
LightOn = AbsTime(1, 10, 0)
```

After the given time has elapsed, the light goes on and stays on.

```
If AbsTime(1, 1, 0);
[
   LightOn = !LightOn;
]
```

The light will switch on and off every second since the IF statement resets the AbsTime function every time it becomes true. One second later, it becomes true again.

```
LightOn = Latch(AbsTime(1, 2, 0), AbsTime(1, 2, 1))
```

This is functionally equivalent to the previous example. The light flashes on and off each second.

```
If TimeOut(1, 1);
[
   J++;
]
```

J will increment once per second, since the Timeout is reset each time it becomes true.

```
Z = TimeOut(1, 1);
If Z;
[
   J++;
]
```

J will increment continuously, as fast as your computer will allow.
When combining function calls and other operations in an action trigger, use care to follow the rules of operator precedence to avoid unexpected results.

Functions That Are Automatically Reset:

- AbsTime
- Change
- DeadBand
- Edge
- Intgr
- Latch
- MatchKeys
- Now
- Pick
- RTimeOut
- TimeOut
- Toggle

- Save
- Watch
- WatchArray
- WinMatchKeys

(*) Note: TimeArrived does not automatically reset.

Considerations for Graphics Functions

While it is generally the case that the order of statements within in state is largely irrelevant, this is not the case for graphics functions. Each item is added to the screen in turn (layered) according to its position within the state.

Focus

Each graphic function that can receive keyboard focus has a parameter called a FocusID. The input focus determines what graphic control receives keyboard input. Like Microsoft Windows™, only one graphic control can receive keyboard input at a time. Each graphic item changes its display to indicate it has the focus. For example, a button shows a dashed line around its label. Pressing the return key while an item has the focus is the equivalent of clicking the mouse on it with the correct button combination (specified in its Button parameter).

The focus number is usually (but not necessarily) unique. The function, NextFocusID can be used to automatically set the focus to the next item with a certain focus ID number; FocusID returns the focus ID number of the item that has focus.

Note that the above refers to keyboard focus. If several objects, such as buttons, can react to a mouse-click, and if those objects overlap on the screen, then that click will be used by all the overlapping objects.

Window Coordinates

In VTScada, the screen is laid out in an x-y grid with the x-axis horizontal and the y-axis vertical. When VTScada is first started, the upper left corner of the screen is the origin (where both x and y are 0). X

represents the number of pixels (dots) from the left side of the window. Similarly, y represents the number of pixels (dots) from the top side of the window.
Note that there is both a function that can provide the current display resolution (VStatus), and a statement that can change the world coordinate limits (Coordinates).
Additionally, the VTScada Coordinates utility enables you to precisely determine the horizontal and vertical coordinates of the mouse pointer within any VTScada window (page or dialog). For further information on the Coordinates utility, please refer to Coordinates Application.

Threading

VTScada provides the ability to have multiple control threads that share a single address space, but appear to behave as if they were separate processes (i.e. the processing time is divided equally among threads of the same priority). This means that those statements that are deemed to be threaded, and any modules that are called in their own thread will not block each other waiting for their turn to execute, but will instead share the processor's time, giving the appearance of simultaneous execution.
The chapter "Function Usage in States, Scripts, and Threading" lists which functions are threaded. In cases where two versions of the same function exist, one being threaded and the other not, the threaded version will generally have a "T" appended to its name, such as in the case of Get and TGet.
Great care must be exercised in using threaded functions or in launching a module in its own thread by using the Thread statement (this is discussed further in the section on modules). Since each thread executes independently of the others, except for its shared memory space, no assumptions may be made as to when the statement is finished execution and its resultant variable assigned a value. Any variables set by threaded statements must be checked for validity before proceeding to

use them. If order of execution is important and the task to be performed is not an overly long and arduous one, it may be better to use the non-threaded version of a function. For example, if you wish to retrieve ten thousand records from a file and don't want your entire application to be suspended while the data retrieval is happening, using a TGet is probably appropriate. If on the other hand you have only two records to retrieve (assuming that they don't each contain a thousand fields) you may find that Get is more appropriate, since any processing of the data can be included in the same script that executes the Get and there will not be any time wasted in creating and destroying the thread in which the function operates.
It should be noted that VTScada provides utilities to assist you in troubleshooting threading in your applications. The Thread List supplies a list of the separate threads of execution for which VTScada is responsible within a local application.

Operators in Statements

Operators are symbols used to perform an operation, comparison, or mathematical function (such as addition or subtraction). Operands are variable names or expressions that are being compared or that a mathematical function is being performed upon.
Some operators are used in expressions by placing the symbol for the operator between two operands. For example:

```
A + B
```

The operands A and B are variable names, but they could also have been expressions. The + operator is placed between the two operands and means that the expression A + B will return the value of the sum of the values of the variables A and B.
Several operators follow a slightly different rule. The logical NOT (~ or !), unary minus (-), preincrement (++), predecrement (--), pointer dereference (*), and address of (&) precede their operands.

Operators may be combined to form more complex expressions, such as :

```
A + B * 5 / C <= 11.5
```

Related Information:

...Operator Priority in Statements

...List of VTScada Operators

... Boolean Logic Operators

...Scope Resolution Operators

Operator Priority in Statements

The order in which the operators are executed is significant. Consider the expression:

```
1 + 2 * 3
```

The value of this expression depends upon whether the addition or multiplication is done first. If the addition is done first, the result is 9; otherwise, it is 7. To resolve this type of ambiguity, VTScada assigns priorities to operators - operators with higher priority are done first. Multiplication has a higher priority than addition, so in the previous example the correct result is 7. If the addition were intended to be done before the multiplication, the expression could be written as:

```
(1 + 2) * 3
```

Parentheses ( ) force the expression within them to be done first. The following two expressions have the same value:

```
1 + 2 * 3
1 + (2 * 3)
```

Some operators have equivalent priorities, such as addition and subtraction. In these cases the evaluation is done starting with the left-most operator. For example:

```
1 - 2 + 4 * 5
```

could be equally written :

```
(1 - 2) + (4 * 5)
```

A detailed list of all VTScada operators and their priority levels is given in the List of VTScada Operators section. If there is any doubt as to whether or not parentheses are required, you should include them. There is no penalty to pay in terms of speed or memory requirements for placing redundant parentheses in an expression.

Note: When doing comparisons between two operands of different types, the second operand is always cast to the type of the first. This can cause differing comparison results depending on the order of the operands.

List of VTScada Operators

The following is a list of all available operators and their order of execution in any statement (priority). Following the table of operators are detailed descriptions of each operator, including their usage and examples.

Note: When doing comparisons between two operands of different types, the second operand is always cast to the type of the first. This can cause differing comparison results depending on the order of the operands.

Description	Symbol	Priority

Parentheses
Use to force operations to happen in a given order. Operations within the parentheses will be done before operations outside.
`X = 2 + 3 * 4; { x will be 14 }`
`X = (2 + 3) * 4; { x will be 20 }`
	( )	1

Scope Resolution: dot and backslash The dot scope resolution operator (.) allows access only to variables and modules within the current scope. If there is no match within the current scope, Invalid will be returned. The backslash scope resolution operator (\ ), allows access to variables and modules outside the current scope. This scope resolution operator must be used with an object value, and a member name (either a variable or module name).	\	2

Array Index Follows the array variable name, and specifies which element within the array is to be used.	[ ]	3
Segment/Offset This returns a number which is the real mode address with the segment before the @ and the offset after the @. This value can be used in functions such as MemIn and MemOut.	@	4
Logical NOT, ~, ! This is the logical NOT operator. If the expression following the ~ is true (non-0), then the function returns false (0). If the expression following the ~ is false, the function returns true (1).	~ or !	5
Unary Minus This operator returns the negative of the numeric expression to the right of it. For example, -X takes the negative of the value of the X variable. It uses the same symbol as the subtraction operator but does not have an argument before it.	-	5

Pointer Dereference
This dereferences a pointer value (see Pointers; that is, it returns the actual value pointed to by the pointer. For example, the following takes the value in var, adds 1, and stores the result to x.
`ptr = &var;`
`x = *ptr + 1;`
The pointer dereference may also be used on the left side of an assignment to change the value pointed at by the pointer, as follows:
`*ptr = x + 1;`
This takes the value in x, adds 1, and stores the result to the variable pointed at by ptr.
This is a powerful tool. It enablesthe destination of an assignment to be changed at runtime. For example, an application may need to set one of 500 variables, depending on some variable q. It would be impractical to write 500 assignment statements inside of 500 separate actions. It would be much simpler to create a 500 element array, with each element pointing at a different variable. Then execute the statement:
`*(ptrArray[q]) = order;`
This assigns the value in order to the variable pointed at by element q
	*	5

of the array of pointers ptrArray. Pointers to new values may be created with the New function.		

Address
This returns a pointer to the operand. For example,
`y = 4;`
`ptr = &y;`
This stores a pointer to y in variable ptr. If ptr is used in any situation requiring a value other than a pointer value (such as a numeric value), the result will be invalid, because ptr is a pointer to a value, not a value itself. For example:
`w = ptr + 1;`
This will cause w to be set invalid; what should have been written was:
`w = *ptr + 1;`
This will set w to 5. Pointers to new values may be created with the New function.
	&	5

Pre-increment and Post-increment
This operator adds one to a number before it is used (pre-increment: ++x) or after it is used (post-increment: x++). For example:
`x = 3;`
`y = ++x;`
Both x and y will receive the value 4 after these statements execute.
`x = 3;`
`y = x++;`
Following these statements, y will have the value, 3 and x will have the value, 4.
	++	5

Pre-decrement and Post-decrement
This operator subtracts one from a number before it is used (pre-decrement: --x) or after it is used (post-decrement: x--). For example:
`x = 3;`
`y = --x;`
Both x and y will receive the value 2 after these statements execute.
`x = 3;`
`y = x--;`
Y will have the value 3, and x will have the value 2 after these statements execute.
	--	5

Multiplication This operator takes two arguments. The returned value is the result of multiplying the numeric expression before the * by the numeric expression after it. If either expression is a valid 0, the result will be 0 regardless of whether or not the other expression is valid.	*	6
Division This operator takes two arguments. The returned value is the result of dividing the numeric expression before the / by the numeric expression after it. If the expression after the / has a value of 0, the result is invalid. It should be stressed that the result of this operation is not necessarily of type integer, even if both arguments were integers.	/	6

Modulus
This operator takes two arguments. The returned value is the remainder when the first argument is divided by the second. For example:
`a = 5 % 2;`
`b = 7.4 % 1.2;`
The values of a and b will be 1 and 0.2 respectively.
	%	6

Addition/Concatenation
This operator takes two arguments. If either argument is a number, Tag or Normalize value, the return value will be the result of adding the two arguments. Otherwise, the return value will be a string of the first argument concatenated with the second argument.
`a = "Bob Smith";`
`b = "Operator " + a + " has logged on";`
In the preceding examples, b will be equal to the string "Operator Bob Smith has logged on".
	+	7

Subtraction This operator takes two arguments. The returned value is the result of subtracting the numeric expression after the minus sign from the numeric expression before it.	-	7
Right Shift This operator works on numeric values only and shifts the bits in the first operand right by the number specified by the second operand; the appropriate number of zeroes go into the bits vacated to the left side of the value. For example:	>>	8

`x = 0b01100111 >> 3;` The value of x will be 0b00001100.		
Left Shift This operator works on numeric values only and shifts the bits in the first operand left by the number specified by the second operand; the appropriate number of zeroes go into the bits vacated to the right side of the value. For example: `x = 0b1100111 << 3;` The value of x will be 0b1100111000. (note: the value is stored in a 32 bit integer - by convention, zeroes to the left of the highest value 1 are not shown)	<<	8
Less Than This operator returns true (1) if the first argument is strictly less than the second, otherwise it returns false (0). If both arguments are text values, the operator returns true if the first argument is alphabetically lower than the second.	<	9
Less Than or Equal To The same as Less Than, but will also return true (1) if the two operands are equivalent.	<= or =<	9
Greater Than This operator returns true (1) if the first argument is strictly greater than the second, otherwise it returns false (0). If both arguments are text values, the operator returns true if the first argument is alphabetically higher than the second.	>	9
Greater Than or Equal To The same as Greater Than, but will also return true (1) if the two operands are equivalent.	>= or =>	9
Equal To This operator returns true (1) if the two operands are equivalent.	==	10
Not Equal To This operator returns true (1) if the two operands are not equivalent.	<> or >< or !=	10
Exclusive OR (XOR, ^)	^	11

This returns the 32-bit, bitwise exclusive OR of the two operands. If either operand (but not both) is true (non-0), the result is true(1); if both are true or both false(0), the result is false.
`p = 0 ^ 0;`
`q = 1 ^ 0;`
`r = 0 ^ 1;`
`s = 1 ^ 1;`
The values of p, q, r, and s are 0, 1, 1, 0 respectively.		

Logical and Bitwise and operations: AND, &, &&
Both the & and the && operators(*) take two arguments and perform the logical AND function upon them. If both arguments are true (non-0), the operator returns a true value; otherwise, a false value (0) is returned. If either argument is a valid false, the function returns false regardless of whether or not the other argument is valid.
The AND operator performs a bitwise comparison (32-bit).
VTScada does not use short-circuit evaluation. Both parts of the condition will always be checked (and evaluated if required).
`If a && b;`
`[`
`  c = d && Sqrt(e - 9);`
`]`
In the above example, the script will be only be executed if both a and b have non-zero values.
(*) In practice, the & operator is not used, avoiding confusion with the C/C++ bitwise comparison operator.
	& or &&	12

Logical and bitwise or: OR, |, || The | and || operators(*) take two arguments and perform the logical OR function upon them. If either argument is true (non-0), the operator returns a true (1) value. If both arguments are false(0), the operator returns a false value(0). If either argument is a valid true, the function returns true regardless of whether or not the other argument is valid. The OR operator performs a 32-bit, bitwise comparison. VTScada does not use short-circuit evaluation. Both parts of the condition will always be checked (and evaluated if required). (*) In practice, the | operator is not used, avoiding confusion with the C/C++ bitwise comparison operator.	| or ||	13
If Else Inline If-Else.	? :	14

This operator takes three arguments; if the first expression or condition evaluates to true (1), the second argument's value is returned, otherwise the third argument's value is returned. The second and third arguments need not have return values, but can be statements with one or more actions to perform. If no return value exists, invalid is returned.
`ZBar(10, 200, 60, 10, Scope(\VTSDB, "ThePort\TheDriver-\ReadVal")\value > 70 ? 12 { red } : 10 { green });`
In this example, color of the bar drawn on the screen will be based on a tag named ReadVal. (Child of "ThePort\TheDriver") When the value is greater than 70, the bar will be red, when it is equal to or less than 70, the bar will be green.
Note that the ? operator has a lower precedence than most other operators, and therefore expressions using it should be enclosed in parentheses when used in combination with other operators.
For example:
`WhileLoop( A < B || SubroutineCall() ? Valid(x) : Valid(y),`
`...`
`)`
... would evaluate as:
`WhileLoop( (A < B || SubroutineCall()) ? Valid(x) : Valid(y),`
`...`
`)`
...whereas:
`WhileLoop( A < B || (SubroutineCall() ? Valid(x) : Valid(y) ),`
`...`
`)`
...is probably what was intended.		

Assignment Assigns the value of the following constant, variable or operand to the variable name that precedes the operator. X = 3; assigns the value 3 to X.	=	15
Add Equals Adds the value of the following operand to the preceding variable. X+= Y is equivalent to X = X + Y.	+=	15
Subtract Equals	-=	15

Subtracts the value of the following operand from the preceding variable. X-= Y is equivalent to X = X - Y.		
Multiply Equal Changes the value of the preceding variable by multiplying it by the following operand. X*= Y is equivalent to X = X * Y.	*=	15
Divide Equals Changes the value of the preceding variable by dividing it by the following operand. X/= Y is equivalent to X = X / Y.	/=	15
Modulus Equals Changes the value of the preceding variable by applying the following operand as its modulus. X%= Y is equivalent to X = X % Y.	%=	15

Boolean Logic Operators

With regards to state logic and the VTScada scripting language, any expression that contains an Invalid, whether it be in combination with the "+", "-", "*", "/", "&&", or "||" operators (see List of VTScada Operators), is expected to return Invalid unless PickValid is used. There is one exception to this rule; that is, if an expression contains an Invalid and a "0", then the result will be a "0", as Boolean logic dictates that anything anded with a "0" is always a "0". The underlying principle is to return a valid result whenever possible.
For example:

```
Invalid && 0 = 0
Invalid && 1 = Invalid
Invalid || 1 = 1
```

Following are some of the common operators and the result when they used in Boolean expressions.
&& (AND)
Two or more items must agree (i.e. must be evaluated to the same result) in order for the expression to be true.

```
1 && 1 = TRUE
0 && 1 = FALSE
1 && 0 = FALSE
0 && 0 = FALSE
1 && 1 && 1 = TRUE
1 && 1 && 0 = FALSE
```

|| (OR)

One, both, or more items must agree. If both inputs are false, the result is false.

```
1 || 1 = TRUE
0 || 1 = TRUE
0 || 0 = FALSE
```

! (NOT)

Reverses the input. If true is input, the result is false; if false is input, the result is true.

```
!1 would evaluate to FALSE
!0 would evaluate to TRUE
```

^ (XOR - eXclusive OR)

Only one input may be true; if both inputs are true, the entire result is false.

```
1 ^ 1 = FALSE
1 ^ 0 = TRUE
0 ^ 1 = TRUE
0 ^ 0 = FALSE
```

Value Types and Storage

VTScada uses many different types of value: integers, floats, text, dictionaries, etc. VTScada is not a hard-typed language. All variables and constants used in a module must be declared, but the intended value type is not part of the declaration and different value types may be assigned to a variable as the module runs.

Variables and constants are declared at the beginning of a module, within one set of square brackets. This section will always be immediately after the parameter list, which is within round parentheses.

```
CustomControl          { module name }
(
   parm1                        { a parameter to the module };
)
[                               { start of variable and constant declar-
ation }
   TotalCount                         { variable declaration
};
   CurrentCount         = 0           { variable declared with an
```

```
initial value    };
   Constant #WarningMsg = "Danger"  { constant declared
with a text value        };
]
```

By convention, constants are declared with a leading "#", but this is only for the sake of convention. The hash mark has no functional significance.

Default Values

Every instance of a variable starts with a value. This value may default to invalid, or it may be declared to have a numeric or text value upon creation. Every time an instance of that variable is created, it will begin existence with its default value. A default value is specified by placing a = after the variable name followed by the desired default value.
Persistent variables will only set their default value if there is no .VAL file containing the persistent value.
The following table provides a list and short description of each of the value types commonly used in code. Where more explanation is required, there are links to relevant topics. From time to time, you may also need to refer to the more technical, table of VTS Value Types.

Common Value Types for Coding

Numbers

Numbers are stored internally in the most efficient form for the value provided. In all cases, VTScada is able to handle double-precision, floating point numbers. All arithmetic is done with the precision of 8-byte IEEE floating point numbers, regardless of the values provided. The legal value range of values ±10^307, and a precision of approximately 15 decimal places.

Use a minus sign to indicate negative numbers. A plus sign may not be used in a number. The following formats may be used as needed:

Format	Example	Description
Scientific notation	-23.5e5	Place the letter "e" or "E" after the number, followed by the number to use for the power of 10. For example, 1.23E3 is the

		same as 1230.0 and 1.23E-3 is the same as 0.00123.
Binary notation	0b1100	Binary integers may be specified using the 0b or 0B prefix and up to 32 digits. Each digit is either a 0 or a 1.
Octal notation	014	Octal integers may be specified using a zero prefix and up to 11 digits. Each digit must be in the range 0 to 7.
Hexadecimal format	0xC	Hexadecimal integers may be specified using the 0x prefix and up to 8 digits. Each digit must be a number or a letter in the range A to F (upper or lower case).

Text / String

Text values (also called "text buffers" or "strings") are a series of ASCII characters. Text values can hold up to approximately 65,500 characters (just less than 64k characters).
A text value may not hold the ASCII 0 value. To store quotation marks in a text variable or constant, use double-quotes:

```
Quote = """We lived for days on nothing but food and
water."" W.C. Fields";
```

The term, "null string" refers to a text value having zero characters.

Logical Values (also, "status value" or "Boolean")

A logical value is a "true" or "false" value. A false value is indicated by the number "0" or the function, FALSE. A true value is indicated by any valid number other than 0, (typically a "1") or by the function, TRUE.
In general, it is better coding practice to use the functions TRUE and FALSE than 1 and 0.

Pointers

A pointer is a variable that holds a memory address. A

pointer references or "points to" a value in a specific memory location.

A variable that points to an address does not have the properties of the value type in that address and cannot be converted to another value type.

A pointer is made to reference the memory address of a variable by using the address operator "&". For example:

```
y = 4;
ptr = &y;
```

Before the data referenced by a pointer can be used, the pointer must be dereferenced.

Streams

Stream values are a complex type, which refer to a read/write stream. Streams are a very convenient way of performing formatted input and output. Examples of streams include buffer streams attached to text values, file streams attached to disk files, and pipe streams attached to operating system pipes. There are a wide variety of functions related to streams.

Most streams contain a position pointer that indicates where in the stream the next read or write will take place. This pointer is automatically positioned after a read or write to the stream and may be positioned by the Seek function, which also returns the current position in characters from the beginning of the stream.

Streams also offer faster file access than FRead and FWrite, which open and close the file after every access, because a stream leaves a file open. The downside is that if the computer is shut off or loses power before a CloseStream closes the file stream, the opened file will be corrupted.

Object Variables

The object type is used to access public members of a module instance. Each object variable references exactly one module instance. Programmers experienced with high level languages may

identify the object type roughly with a pointer type from Pascal or C (the pointer points to a particular copy of a module). Every active module call creates a new separate copy, or instance, of that module and its member variables. Each module instance operates independently of the other instances.

Graphic Variables

VTScada uses a set of variable types for graphics that are unique to VTScada. These value types are used by all layered graphics functions, and by the functions which create them. All of these values are valid only so long as the function that created them is active.
This set of value types includes: Point, Vertex, Path, Normalize, Rotate, and Trajectory Variables

Array Variables

Arrays are a special type that enables a group of values to be kept under one variable name. The individual elements of the array are sequentially numbered. Each of the elements of the array can be used as a completely separate variable with its own independent value that may have its own type.
There are two types of array: static and dynamic, which refer to how the array is declared, not the values within it.

Dictionaries

A dictionary is flexible data structure, providing functionality very much like a database. It is a named structure, holding a flexible set of name-value pairs , which may have a value itself. Dictionaries may hold other dictionaries.

Structures

Structures allow you to organize information to increase the overall clarity of your code. Much like a structure in C, these are collections of variables and their values, placed under one name.

Related Information:

...ASCII Constants

...Value Type Conversions

...Invalid Values

...Using Arrays

...Using Pointers

...Dictionaries

...Meta Data

...Structures

...Variable Storage, Retention, Access

...Variable Class Definitions

...VTScada Value Types - Numeric Reference

Value Type Conversions

Occasionally, data may be the wrong type. For example, an addition may be performed on two variables, one containing a number, the other containing a text value. VTScada handles this situation by performing a conversion of the unexpected data to the type of data expectd.

```
A = 5;
B = "4";
C = A + B;
```

In this example, two numbers are expected. One variable is already a number; the other variable containing text is converted by VTScada to a number. This is done by reading an ASCII number from the text. If successful, the converted number is used; if unsuccessful, an invalid value is returned. This process is referred to as "type conversion".

Note: Note When two variables of different type are used in an expression, the second operand will always be cast to the type of the first. This can cause radically differing results depending on the order of the operands.

Table of VTS Value and Type Conversions

Convert From	Convert To	Condition of Original Value	Returned Value
Code Pointer	Module	Valid value	Valid value
	Module State	Valid value	Valid value

Module State	Statement	Valid value	Valid value
	Object	Valid value	Valid value
	Text	Valid value	Name of module
Double	Long	Valid value	Valid value
	Short	Valid value	Valid value
	Status	Valid value	Valid value
	Text	Valid value	String representing numeric value
Edit Block	Stream	May only be used in SRead with % option, or in StrLen	Valid value
Long	Double	Valid value	Valid value
	Short	Valid value	Valid value
	Status	Valid value	Valid value
	Text	Valid value	String representing numeric value
Module	Text	Valid value	Name of module
Module State	Module	Valid value	Valid value
	Text	Valid value	Name of module
Module State Statement	Module	Valid value	Valid value
	Module State	Valid value	Valid value
	Text	Valid value	Name of module
Normalize	Double	Valid value	The scaled value
	Long	In the range of -2 147 483 648 to 2 147 483 647	The value
	Short	In the range of -32 767 to 32 767	The value
		Outside of range	Invalid
	Status	0	0 (false)

		Non-0	1 (true)
	Text	Valid value	String representing scaled value
Object	Module State	Valid value	Module and state in which that object exists
	Statement	Valid value	Module state and statement number that object is executing
	Text	Valid value	Name of the module of which that object is an instance
Short	Double	Valid value	Valid value
	Long	Valid value	Valid value
	Status	Valid value	Valid value
	Text	Valid value	String representing numeric value
Status	Double	Unconnected socket	1
		Connected socket	Invalid
	Long	Unconnected socket	1
		Connected socket	Invalid
	Short	Unconnected socket	1
		Connected socket	Invalid
	Status	Unconnected socket	1
		Connected socket	Invalid
	Text	Unconnected socket	"1"
		Connected socket	String of stream contents
Tag	Double	Valid value	The scaled value
	Long	In the range of -2 147 483 648 to 2 147 483 647	The value
		Outside the range	Invalid
	Short	In the range of -32 767 to 32 767	The value
		Outside the range	Invalid

	Status	0	0 (false)
		Non-0	1 (true)
	Text	Valid value	String representing scaled value
Text	Double	Number string	Number in string
	Long	Number string	Number in string
	Short	Number string	Number in string
	Status	Non-0 number string	0 (false)
		0 number string	1 (true)
Variable	Module	Module variable	The module in which the module variable resides
	Text	Valid value	Name of variable

Invalid Values

In addition to having normal valid values, variables can also be invalid. An invalid value is an unknown value that is distinct from all other values; they guard the system from performing control action based upon erroneous or bad information. Any variable may have an invalid value. Any calculation that uses an invalid value produces an invalid result (with a few exceptions). Invalid results are not written to output devices. In addition, any graphic statements that contain invalid variables will not produce any output on the screen. In general, any function that uses an invalid value will have no effect and behave as if it did not exist.

Invalid values may result from variables that are not set to any given value - this is true when VTScada is first starting up. To avoid this situation, programmers use the PickValid() function to provide a default value to expressions until the variable has a valid value.

Invalid values can also result from setting a variable's value in two or more different statements at the same time; in this situation it is uncertain which value is correct, so the variable will be invalid even if both values are the same. This is known as a "double set". An array reference that has one of its subscripts out of range will have an invalid value. Some programmable controllers flag inputs when they are out of range - when

these values are read by VTScada, they are declared to be invalid. Division by 0, taking the square root or logarithm of a negative number, and other illegal mathematical operations result in invalid results. An invalid value can even be purposely set as such by using the Invalid function. Because calculations using invalid values produce invalid results, invalid values can propagate through the system. The setting of a variable to invalid will cause all other variables that depend upon its value either directly or indirectly to become invalid.

Using Arrays

An array is a value type used to group a set of values under one variable name. The individual elements of the array are sequentially numbered. Each of the elements of the array can be used as a completely separate variable with its own independent value that may have its own type.

Note: Arrays may be declared either as static or dynamic. If a function specifies one type or the other as a parameter, it is essential that you use only that type.
In all cases where a static array is not specifically required, use a dynamic array. In practice, this means that the use of a static array is uncommon.

A dynamic array is created by first, declaring the name:

```
[
   NameArray      { to be used as a one-dimensional array };
   AddressArray   { to be a two-dimensional array         };
]
```

A script block is then used to define the array size, using the New() function.

```
Init
[
   IF 1 Main;
   [
      NameArray = New(10);
      AddressArray = New(2, 10);
   ]
]
```

If you need to initialize the array after creating it, use an ArrayOp() function.
Static arrays are created, and optionally initialized when they are declared:

```
[
   aStaticArray[10] = 0;
   AnotherStatic[2][10];
]
```

When referring to a particular element of an array, the number enclosed in square brackets [ ] is called the "subscript". You may use a variable, an expression or a function as the subscript in your code, in order to manipulate an array or dynamically access different elements as operating conditions change.

Related Information:

...Multidimensional Arrays

...Mismatched Array Dimensions

...Array Processing Functions

...Comparison of Static and Dynamic Arrays

...Array

Multidimensional Arrays

An array that contains arrays in its elements is defined as follows:

```
Data[5][10];
```

This declaration may be read as "Data is an array of 5 elements each of which is an array of 10 values." This is a multi-dimensioned array. Each dimension must appear as a number enclosed in square brackets [ ] that follows the variable name, but you may also define starting and ending elements. For example, the following declaration is for a two-dimensional array whose valid elements range from Data[10][-1] to Data[20][6]:

```
Data[10, 20][-1, 6];
```

The left-most subscript is the "first" or "lowest" dimension of the array. The right-most subscript is the "last" or "highest" dimension.

For functions and statements that take arrays as parameters, the second-to-last subscript specifies the sub-array to use. For arrays with only one dimension, there is no ambiguity. For example, consider the statement that begins:

```
Plot(DataX[2][3], 10, ...
```

The values in the array DataX[2][3] to DataX[2][12] will be plotted. The last subscript is the only subscript that varies. Any preceding subscripts remain fixed for the function or statement. This rule means that it is not possible to directly plot the 10 values contained in DataX[0][0] to DataX [9][0], however, it is possible to plot DataX[0][0] to DataX[0][9]. This principal applies for all functions and statements that require arrays as parameters, but only when there is more than one array dimension.

For functions that return multidimensional arrays, data may not be stored in the manner that you would expect if you are familiar with other programming languages. If unsure, use the Source Debugger to examine the array structure. Consider the following example, where the third mode of ListKeys() is used to obtain an array of keys and values from a dictionary The stored data might be represented in some languages as the following array:

Given the code:

```
IF Watch(1);
[
  X = Dictionary(0, 5);
  X["A"] = 42;
  X["B"] = 86;
  X["C"] = 99;
  Buf = ListKeys( X, 1, 3);
]
```

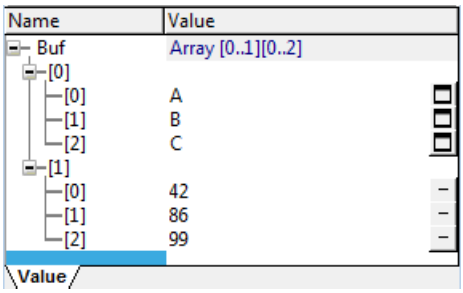
Some programmers might expect the return array to contain 3 rows of 2 columns:

A 42

B 86

C 99

but in fact, Buff will contain two arrays of three values:



The reason for this behavior is efficiency. It is faster to build the return array as shown than otherwise. If in doubt, always create a test case to examine using the Source Debugger.

Related Information:

...Mismatched Array Dimensions

...Array Processing Functions

...Comparison of Static and Dynamic Arrays

Mismatched Array Dimensions

If the number of defined array dimensions does not match the number of array dimensions specified in an array reference, the evaluation proceeds anyway, following certain rules. Specified array dimensions are matched to defined array dimensions starting from the right and working towards the left. If more array dimensions were specified than were defined, that array expression is called "over specified". If more array dimensions were defined than were specified, that array expression is called "under specified".

Over-specified array expressions will return an invalid value with one exception - if all leading over specified dimensions are zero, it will return element 0 of the specified array. For example:

```
Data[2][3];
Q = 4 + 3 * Data[X][1][2];
```

This will work if X is zero. Otherwise, Q will be set invalid. For under-specified arrays, missing leading dimensions will default to zero. In the previous example, consider the statement:

```
V = 4 + 3 * Data[1];
```

This expression would use Data[1][0].

Related Information:

...Multidimensional Arrays

...Array Processing Functions

...Comparison of Static and Dynamic Arrays

...Array Processing Functions

In data processing, the WhileLoop() and DoLoop() functions certainly have their place; however, when faster performance is required, the VTScada array processing functions are recommended (see "Array Functions").

For example:

```
WhileLoop(I < N;
  Array[I] = 0;
  I++
)
```

This can be replaced by the simpler:

```
ArrayOp1(Array[0], N, 0, 0)
```

If used in code that executes often, the second version will run significantly faster than the first.

Another example involves pre-formatting the data received from an I/O device in order to read that data into arrays. Whenever possible, BuffToArray (and its sister BuffToParm) should be used for this task. In many situations, the data in these return messages is not in the correct format to use directly in the BuffToArray function, and it would appear at first glance that a loop must be used. The data can instead be pre-processed using an array function, and then passed to BuffToArray (an array processing function itself).

If, for example, you are reading four-byte HEX ASCII numbers from a device, but the bytes are received in high to low order (the opposite to what the BuffToArray function expects). Further, all data bytes representing 0 (0X30 HEX ASCII) have the highest order bit (b7) set to 1. The following fragment of code will read these values into an array correctly without requiring a loop.

{ Clear the high order bit of the 0 values element }

```
RcvBuff = Replace(RcvBuff, 0, N * 4, MakeBuff(1, 0x30 + 0x80),
MakeBuff(1,0x30));
```

{ Swap the byte order }

```
RcvBuff = BuffOrder(RcvBuff, 0, 1, 4, N);
```

{ Read the data values }

```
BuffToArray(Array[0], N, RcvBuff, 5, 5, 0);
```

Alternatively, to be even more compact:

```
BuffToArray(Array[0], N,
            BuffOrder(Replace(RcvBuff, 0, N * 4,
                      MakeBuff(1, 0x30 + 0x80),
                      MakeBuff(1, 0x30)), 0, 1, 4, N),
            5, 5, 0);
```

As a third, even faster option, you can remove the MakeBuff functions and replace them with text constants that are set only once.

Note: You should always consult the array functions section whenever you have a coding task that will require extensive looping. In many cases, one or more array processing functions will exist to help to solve the problem. The array functions are listed in "Array Functions".

Related Information:

...Multidimensional Arrays

...Mismatched Array Dimensions

...Comparison of Static and Dynamic Arrays

...Array

Comparison of Static and Dynamic Arrays

A static array variable holds a value of type, array. A dynamic array variable holds a pointer to an array value. Thanks to automatic pointer dereferencing and automatic index padding, there is very little difference in the code that you write to use either type of array. What differences do exist can be found in the following technical points.
It is important to note that use of a static array where a function expects a dynamic array, will result in that function failing to work properly. Use

dynamically allocated arrays for all code unless a function reference explicitly states that a static array is expected for one of the parameters. Static arrays are created in the variable declaration [StaticArray[50]], whereas dynamic arrays are created using the New() function within a script block.

Automatic Pointer Dereferencing

Since dynamic array variables are pointers, one might expect to do a lot of pointer dereferencing when using dynamic arrays. For example, to access the fourth element of a dynamic array pointed to by DynamicArray, one might expect to write the following code:

```
(*DynamicArray)[4] = 0;
```

While this syntax is correct, it is also redundant; whenever VTScada encounters the [ ] index operator, it automatically dereferences the variable being indexed. This makes the explicit dereferencing unnecessary, so that the above code can simply be written as:

```
DynamicArray[4] = 0;
```

By automatically dereferencing an array pointer under these circumstances, VTScada makes accessing an element of a dynamic array look and behave just like accessing an element of a static array.

```
StaticArray[4] = 0;
```

Note: VTScada only performs automatic dereferencing when an element is accessed with the index operator. In no other cases is a dynamic array automatically dereferenced.

Automatic Index Padding

Unlike its automatically dereferenced pointers, VTScada's automatically padded indices makes for array-related code that looks the same, but behaves differently, depending on whether a dynamic array or a static array is used. This mismatch between appearance and workings can easily lead to developer confusion, and thus warrants special attention. VTScada applies automatic index padding whenever an array reference is under-specified. In turn, an array reference is considered under-specified if two criteria are met:

1. The array reference must involve an array value, as opposed to a pointer to an array value. This is almost the same as saying the array reference must involve a static array variable, as opposed to a dynamic array variable, with two exceptions:

    a. Whenever a dynamic array variable is explicitly dereferenced (*DynamicArray), the result is an array value.
    b. Similarly, whenever a dynamic array is referenced with the [ ] operator, VTScada automatically dereferences the array pointer, and the result is an array value.

2. An array reference must include fewer [ ] operators than the referenced array has dimensions.

Given a two-dimensional dynamic array variable named, DynamicArray, and a two-dimensional static array variable named, StaticArray, the following code shows under-specification:

```
1. *DynamicArray;     { Under-specified }
2. DynamicArray;   { Not under-specified, not an array value }
3. DynamicArray[2];    { Under-specified, automatic dereferencing }
4. DynamicArray[1][3]; { Not under-specified }
```

```
5. StaticArray;       { Under-specified }
6. &StaticArray;      { Not under-specified, not an array value }
7. StaticArray[2];   { Under-specified }
8. StaticArray[1][3]; { Not under-specified }
```

Lines 2 and 5 show the inconspicuous difference between dynamic and static arrays: VTScada considers line 2 fully specified, but considers line 5 under-specified.

Whenever VTScada encounters an under-specified array reference, it automatically pads the least significant, under-specified indices with zero indices. Revisiting the example above, the array references are padded as described in the comments in the following code.

```
x = *DynamicArray;     { x = DynamicArray[0][0] }
x = DynamicArray; { x = DynamicArray. See Variable Assignment }
x = DynamicArray[2];    { x = DynamicArray[2][0] }
x = DynamicArray[1][3]; { x = DynamicArray[1][3] }
```

```
x = StaticArray; { x = StaticArray[0][0] }
x = &StaticArray;     { x = StaticArray. See Variable Assignment }
x = StaticArray[2];    { x = StaticArray[2][0] }
x = StaticArray[1][3]; { x = StaticArray[1][3] }
```

Value Assignment - Static versus Dynamic Arrays

It is not possible to assign a value directly to (or from) a static array variable, and VTScada's automatic index padding ensures that this does not happen. If it ever appears that a static array is participating in an assignment, automatic index padding works to make sure that only an element of the array is involved. The following code shows the automatic index padding at work, with equivalent assignments in comments.

```
[
   AStaticArray[3] = 1;
   BStaticArray[5][10] = 2;
   SimpleVar;
]
Init [
   If 1 Main;
   [
      SimpleVar = AStaticArray;       { SimpleVar = AStaticArray[0] }
      SimpleVar = BStaticArray[4];    { SimpleVar = BStaticArray[4]
[0] }
      SimpleVar = BStaticArray[3][1]; { SimpleVar = BStaticArray[3]
[1] }
      SimpleVar = &AStaticArray;      { SimpleVar = &AStaticArray }

      AStaticArray = "super";         { AStaticArray[0] = "super" }
      BStaticArray[4] = 3;            { BStaticArray[4][0] = 3 }
      BStaticArray[3][1] = 4;         { BStaticArray[3][1] = 4 }
      &AStaticArray = 0x00008000;     { Error! Not an lvalue }
      AStaticArray = BStaticArray;    {AStaticArray[0] = BStaticArray
[0][0] }
   ]
]
```

Dynamic-array variables are declared no differently than any other VTScada variable. This similarity carries over to assignment, where assigning a value to (or from) a dynamic-array variable is no different than assigning a value to (or from) any other VTScada variable. Dynamic-array variables can be assigned any scalar value, including another dynamic-array variable (a pointer to an array). The following code shows these assignments.

```
If 1 Main;
[
   ADynamicArray = 5;                 { a scalar assignment }
   ADynamicArray = New(3, 2);         { a new dynamic array }
   BDynamicArray = ADynamicArray[1];  { under-specified. ADy-
namicArray[1][0] is assigned instead }
   BDynamicArray = "super";           { another scalar assignment }
   CDynamicArray = New(10);           { a new dynamic array }
```

```
   DDynamicArray = CDynamicArray;     { one dynamic array to another
}
   SimpleVar = DDynamicArray;         { a dynamic array to a simple
variable }
]
```

When a dynamic-array variable is assigned to another variable, as in the last two assignments of the previous example, only the pointer is assigned, and no arrays are copied: in the previous example, CDynamicArray, DDynamicArray, and SimpleVar all point to the same dynamically allocated ten-element array.

Note than during these assignments, neither of VTScada's automatic array-related features was invoked: no [] index operators were present and none of the array references were under-specified.

Passing an Array to a Module

Arguments to a Launched Module (or subroutine) are passed by value (i.e. the value of the argument is copied, or assigned, to the module's parameter). For this reason, passing arguments to a launched module is very similar to assigning a value to a variable. It should be no surprise, then, that the rules that govern passing arguments to launched modules are identical to those that govern variable assignment: VTScada uses automatic index padding to ensure that only an element of a static array is passed to a launched module, while passing dynamic arrays to launched modules is less restricted.

When arguments are passed to a Called Module, they are passed by reference (i.e. a called module's parameter is made to refer the argument, and no copy is made). This is the only case when a static-array reference is not index padded by VTScada. When a static array is passed to a called module, the called module's parameter is made to refer to the entire static array, and not just to one particular array element. Consequently, a called module can change the elements of a static-array argument, and any changes will appear outside of the module.

Similarly, when a dynamic array is passed to a called module, the array pointer is not copied to the module parameter, but rather the called module's parameter is made to refer to the array pointer (think of a pointer

to a pointer). In this way, a called module can change not only the elements in, but also the size and dimensions of, a dynamic-array argument, and any changes will appear outside of the module.

Related Information:

...Multidimensional Arrays

...Mismatched Array Dimensions

...Array Processing Functions

Using Pointers

Many VTScada functions require or return pointers. A pointer variable is created by using the & notation in front of an existing variable name. For example

```
Y = 7;   { "Y" is a the name of a variable, within which is stored
the number "7" }
X = &Y;  { X is the name of a variable, within which is stored a
pointer to the memory address of Y }
```

Pointers must be dereferenced before the data they point to can be used. Dereferencing means accessing the value stored in the assigned memory location. A pointer can be dereferenced by placing an asterisk before it. For example:

```
*ptr
```

The value stored in the location referenced by "ptr" is now available for use.
If you were to use the variable "Ptr" in an expression, you would have to dereference it within the expression. For example:

```
x = *ptr + 1;
```

A dereferenced pointer can be used to store a new value into the location or object at which it is pointing. For example:

```
ptr = &y;
*Ptr = 7;
```

This is equivalent to the expression "y = 7;".

If the pointer "ptr" is used in any expression requiring a value other than a pointer value (for example, a numeric value), the result is invalid, as "ptr" is a pointer to a value, not a value itself. For example:

```
w = ptr + 1
```

In this example, w is set to invalid, as the pointer "ptr" was not dereferenced. VTS will not take this to mean the next memory address. What should have been written is:

```
w = *ptr + 1
```

Pointers and Arrays

Certain VTScada functions, such as New(), return pointers. The New() function is typically used to allocate a dynamic array. For example:

```
ArrayPtr = New(1 { Number of dimensions },
               0 { Starting index },
               10 { Number of elements });
```

Technically, in order to use an element from "ArrayPtr", you must dereference the element:

```
(*ArrayPtr)[0] = 5;
```

Fortunately, the array index operator [ ] automatically performs a pointer dereference operation if the variable before the array index operator has a pointer value. For example, the expression above can also be written as follows to achieve the same result.

```
ArrayPtr[0] = 5;
```

The preferred method is to allow the array index operator to automatically perform the dereferencing of the pointer, as it improves the readability of the statement, requires less memory, and executes faster.

Dictionaries

A dictionary is flexible data structure, providing functionality very much like a database. It is a named structure, holding a flexible set of name-value pairs , which may have a value itself. Dictionaries may hold other dictionaries.

In the following example, MyDictionary has a value of 43 and holds three name-value pairs, one of which is a dictionary having the value, "Greetings" and itself holding two name-value pairs.

```
MyDictionary = Dictionary();    { The dictionary is created }

RootValue(MyDictionary) = 43; { A root value is assigned  }

MyDictionary["ValueA"]  = 5;  { The first stored value is created }
MyDictionary["ValueB"]  = 10; { The second stored value }
MyDictionary["YourDictionary"] = Dictionary(); { The third stored
value }

      { Further definition of the third stored value }
RootValue(MyDictionary["YourDictionary"]) = "Greetings";
MyDictionary["YourDictionary"]["ValueA"]  = "Good";
MyDictionary["YourDictionary"]["ValueB"]  = "Morning";
```

Dictionaries have the following basic attributes:

- Information is stored in the form of key / value pairs.
- Information is automatically sorted by key, resulting in fast searching and retrieval.
- Key / value pairs can be added and removed efficiently and without limit.
- Dictionaries can have root values: a value, tied to the dictionary itself rather than to a key within the dictionary. This is optional, but it should be noted that most operators and functions will treat dictionaries as if the dictionary was simply its root value.
- The key will be stored as text. A number may be used, but note that this will be cast to a text value.
- Any data type, including another dictionary, may be used in a value. You can use this to build complex data structures.

Within a dictionary, all keys will always be unique by definition. Attempting to create a duplicate key by assigning a new value to a key with a non-unique name will simply result in the original key being assigned the new data value. Dictionaries may be defined such that their keys are case sensitive (in which case "a" will come after "Z") or they may be non case sensitive. Unless otherwise specified, they will not be case sensitive.

Related Information:

...Creating a Dictionary

... Dictionary Operations

Creating a Dictionary

Dictionaries can be created by either of the 2 following methods:

Method 1: the Dictionary() function

```
X = Dictionary ([ case ], [ root ]);
```

Both parameters are optional. The Case parameter will default to TRUE, meaning that it is not case sensitive, and the Root parameter will default to Invalid. The root can be any value, including another dictionary. Examples:

```
X = Dictionary();
```

X becomes a reference to a blank, empty dictionary that is not case sensitive

```
X = Dictionary(1, "rate");
```

X becomes a reference to a dictionary having a root value of "rate", that is not case sensitive.

```
X = Dictionary(0, 42);
```

X becomes a reference to a dictionary having a root value of 42, and that is case sensitive.

Note: Whether or not a dictionary contains a root value has an impact on the result that the ValueType() function will return from it.
If there is a root value then, in effect, the dictionary serves to attach metadata to that existing variable and the ValueType() command will return the value type of the root value.
If there is no root value, then the ValueType command will return type 47 from the dictionary.

Method 2: The MetaData command

The command MetaData can be used for *two* different purposes. If used with a parameter that is not a dictionary, the result is to attach metadata to that object (thereby turning it into a dictionary).

```
MetaData(Dictionary,  Key,  [case sensitive]);
```

Example:

```
MetaData(X, "Width", 1) = 5;
```

X becomes a non-case sensitive dictionary, having no root value and possessing one key named "Width" that has the associated value, 5.
More commonly, you would use this to attach extended information to an existing variable as shown in the following example:

```
Y  =  10;
MetaData(Y,  "Area")  =  20;
```

In this example, Y starts as an integer and then becomes a non-case sensitive dictionary having a root value of 10 (the original value) and possessing one key named "Area" which initially has a value of 20.
You can even use this technique to turn an array into a dictionary – something that could not be done using method 2.

```
Z = New(5, 10);
MetaData(Z,  "Rate", 0) = 42.7;
```

Z becomes a case sensitive dictionary having a root value which is the array [5, 10] and possessing one key named "Rate" which has a value of 42.7.

Related Information:

... Dictionary Operations

Related Functions:

... Dictionary

... DictionaryCopy

... DictionaryRemove

... MetaData

Dictionary Operations

Operations involving dictionaries and other variables:

The result of any operation that uses a dictionary as one of the operands and a non-dictionary as the other will always take the root value of the

dictionary as the value to be used for the operation.
Thus, if you use the dictionary from the preceding examples, having a root value of 5, the following would be true:

```
X = Dictionary(0,5); {X is a dictionary with a root value of 5 }
X["A"] = 42; {add a keyed value to the dictionary }
Y = 2;
Z = X * Y; { the root value of X is used for the multiplication }
```

Z now holds the integer value, 10.

Accessing values in a dictionary:

Values within a dictionary can be accessed using an array-like syntax.
The root value is accessed using empty quotation marks.
Examples:
(Using the dictionary X from the preceding example)

```
Y  =  X[""];    { Y now holds the value  5 }
Y  =  X["A"];   { Y now holds the value 42 }
```

Retrieving an array of the keys from a dictionary:

The function, ListKeys will return a one dimensional array of the keys stored in a dictionary

```
RVAL  =  ListKeys ( dictionary );
```

Example:

```
X =  Dictionary();
X["A"] = 15;
X["B"]  = 24;
RVAL = ListKeys ( X );
```

RVAL will now contain a two element array, containing the values "A" and "B".

Retrieving the root value from a dictionary:

The root value of a dictionary can be obtained by either of two methods:

```
Directly:    Y =  X[""];    { where X is a dictionary }
Via a function:Y =  RootValue( X );
```

In general, the result will be identical, except for the case where the root value is another dictionary. In such a case, RootValue will traverse the dictionaries until it finds the first root value that is not another dictionary. (See example 1)

In the case where all the root values are other dictionaries (a circle) then RootValue will return a dictionary, selecting the first root value after the one indicated in the command that points to an earlier value. (See example 2)

Advanced Situation 1 – Dictionaries containing dictionaries:

Given three nested dictionaries
The root of dictionary X is dictionary Y
    The root of dictionary Y is dictionary Z
        The root of Z is the integer 42.
If you retrieve the root of x directly:

```
RVAL =  X[""];
```

Then, RVAL is now dictionary Y
If you were to use the RootValue() function instead:

```
RVAL = RootValue(X);
```

RVAL is now the integer 42

Advanced Situation 2 – Dictionaries with circular links:

Given three dictionaries as follows:
The root of dictionary X is dictionary Y.
    The root of Y is dictionary Z.
        The root of dictionary Z is a link back to dictionary Y.
If you apply the RootValue() function to dictionary X...

```
RVAL = RootValue(X);
```

then RVAL is now dictionary Y, since that is the last root value found in the chain before it looped back to an earlier dictionary.

Assignment operations involving dictionaries

When assigning a dictionary to a variable using the assignment operator (=) , the result is a pass-by-reference, effectively creating an alias for the dictionary rather than a copy.
Example:

```
X = Dictionary(); { create an empty dictionary }
Y = X;            { assign it to Y }
X["A"] = 42;      { create a node in X with key "A" & value 42 }
Y["A"] == 42;     {  TRUE because Y is an alias for X }
```

A function exists to do a pass by value, allowing you to create a copy of a dictionary:

```
RVAL = DictionaryCopy( dictionary,  [deep],  [acyclic],  [lock]);
```

The three optional parameters, deep, acyclic and lock are each Boolean values with a default of FALSE.

- **Deep** If true, all contained dictionaries are copied as well as the dictionary referred to by name. Note that if Deep is set to FALSE, the copied dictionary will not be missing the contained dictionaries – the difference is that in one case, the contained dictionaries are copied as well as the base dictionary, and in the other, the copy of the base dictionary will also include the original, contained dictionaries.
- **Acyclic** If true, cyclic links are removed
- **Lock** If true, all values in the copy will be locked as constants.

Adding and Removing Keys

Keys and values can be added to a dictionary by simply referencing them like so:

```
X = Dictionary(); {  creates a new, empty dictionary }
X["A"]  =  42;   { adds a new key-value pair to dictionary X }
```

Keys and their associated values can be removed from a dictionary using the DictionaryRemove() function as follows:

```
DictionaryRemove(dictionary, key);
```

The given key, specified in the second parameter, and its associated data will be removed from the given dictionary specified in parameter 1.

```
DictionaryRemove has no return value and no optional parameters.
```

Testing whether an object is a dictionary

Most operators and functions will treat a dictionary as if it were simply the variable stored as the root value. Attempting to use the ValueType() command on a dictionary will not work as expected for this reason, since if the dictionary has a root value, then only the value type of the root will

be returned.
The function HasMetaData() will determine whether or not an object is a dictionary:

```
Rval  =  HasMetaData(variable);
```

This function will return TRUE if the variable is a dictionary and FALSE otherwise.

Meta Data

The Metadata() function provides a means of attaching extended information to variables. The concept is inspired by XML and is based on dictionaries.
For example, in XML one might see the following structure, which attaches two attributes (name1 and name2 containing values "x" and "y" respectively) to the object, 2.

```
<tag name1="x" name2="y">
2
</tag>
```

The equivalent in VTS is a dictionary with the root value, 2, and having two name-value pairs, name1 and name2:

```
X = 2;                        { X starts as a simple numeric variable
with the value, 2 }
MetaData(X, "name1") = "x";  { X is now automatically made a dic-
tionary, with the root value of 2. }
MetaData(X, "name2") = "y";  {   Each call to MetaData() adds another
name-value pair to X }
```

Structures

Structures allow you to organize information to increase the overall clarity of your code. Much like a structure in C, these are collections of variables and their values, placed under one name.
The use of a structure is illustrated by the following example:

```
Mod()
[
  Dims  STRUCT[
                LENGTH;
                WIDTH;
                HEIGHT;
        ];
]
```

```
Main
[
   …
]
```

An instance of the structure Dims can now be assigned to a variable as follows:

```
A  =  Dims();
```

You can also assign values to the various nodes in the same statement:

```
A  =  Dims(15, 30, 45);
```

In either, the variable A will now contain an array that is 3 items long and that can be indexed using the keywords Length, Width and Height as defined in the structure. In effect, Length is taken to mean "index 0 of the array stored in A", Width will mean "index 1 of the array stored in A", etc.
The values can now be used as shown in the following examples:

Example 1:

```
Rval  =  A\Width;
```

Rval now holds the value 30.

Example 2:

```
A\Length  =  42;
```

42 is now the value stored in index position 0 of our structure.
You can also use array indexing notation to access the underlying data:

```
A[1] == 30;
```

Structures and Dictionaries

It is interesting to note that structures are based on the concepts of VTScada dictionaries and metadata.
The example above will create a variable named Dims whose actual value is INVALID and whose default is a locked dictionary. The underlying structure of Dims can be visualized as follows:

```
Dims = Invalid   { dictionary with a root value of Invalid }
   Length = 0,   { meta data elements of the dictionary }
```

```
   Width  = 1,
   Height = 2
```

When Dims was instantiated as A in the examples above, A became a one dimensional array containing within it a pointer to Dims. This enables us to reference the contents of A using the keywords Length, Width and Height and get the correct pointers into the array A.
Because of this, it is possible to store additional data of use to the structure within the defining dictionary. In particular, the name of the structure is stored in the dictionary's root. Thus, to identify an unknown structure, you can use the following expression:

```
RootValue(Cast(<struct>, 47);
```

Extending Structures

Structures may be extended as shown in the following example:

```
Y STRUCT [
           A;
           B;
]; { structure 1, named Y }

X:Y STRUCT [
            C;
            D;
]; { structure 2, named X extends Y }

A = X(); { variable A now holds a 4 element array,
           where the elements of the array will be in the order A,
B, C, D }
```

Structures can also be compound data structures, containing other structures as well as more basic data types:

```
Z STRUCT [
           A STRUCT [
                      Q;
                      R;
           ];
           B;
           C STRUCT [
                      S;
           ];
];
```

Variable Storage, Retention, Access

Variable scope through modules and submodules is discussed in the chapter, Module Scope.
Within a VTScada application, you may need to share values between module instances. You may require values to remain in memory while an application re-starts, or you may want to control how values are shared across the servers running an application. These are the topics of this chapter.

Related Information:

...Module Scope

...Variable Class Definitions

...VTScada Value Types - Numeric Reference

...Persisted Variables

...Retained Variables

...Shared Variables

...Saved Variables

...Network Values

...Temporary Variables

...Protected Variables

Persisted Variables

Normally, variable values are kept in RAM, which means that their value will be lost whenever VTScada is stopped such as in the event of a power failure. This is usually not a problem for most values since they will be read in automatically from the plant I/O devices when the program is restarted. However, certain values such as process set points must be maintained, regardless of any interruption. The use of persistent variables solves this problem.
Persistent variables (formerly known as "static" variables in some early versions of VTS) function identically with any variables except that a copy

of their value is kept on disk. All of the persistent variables in a module will be stored in a file with the same name as the module and a .VAL extension. When VTScada is restarted, the persistent values will be read from the .VAL file(s). This means that the value will always be maintained regardless of whether or not the program has been stopped.
Any variable may be defined as being a persistent variable by prefixing its definition with the keyword "Persistent". For arrays, all elements of the array become persistent values if the array itself is defined to be persistent.
The penalty to pay for making a variable persistent is that it requires a substantial amount of time to change its value. It requires a noticeable fraction of a second compared to a fraction of a millisecond for other variables. Therefore, persistent variables should not be used carelessly. They should only be used for values which are only updated occasionally such as operator-entered set points.
It does not require any additional time to reference or use the value in a persistent variable since a copy of its value is also kept in RAM.
Persistent numeric variables are always kept to their full precision on disk, but there is a limit on the number of characters kept on disk for persistent variables which hold text values; the default is 5 characters. This limit may be increased when the variable is defined (up to 65500 characters):
Persistent 15 UserName;
This will cause up to 15 characters of the variable UserName to be saved on disk. Any characters beyond the 15th will simply be discarded. The limit should be kept as small as practical since the use of large limits will increase the file space required as well as the update time for the variables. Persistent text values are always kept with no characters lost while in RAM, so it is only when VTScada is stopped and restarted that the (limited) persistent value is assigned to the variable.
The limit you specify on text values is rounded to the next largest space that will hold that many characters. The space in the file is allocated in chunks of five characters each. For example, if you chose a limit of 42

characters, space would be allowed for Ceil (42 / 5) * 5 = 9 * 5 = 45 characters.
Persistent variables are automatically considered to be shared as well. This cannot be changed by the system designer.
Persistent variables values on disk are erased if the module where they are defined is compiled, or any part of that module is compiled. For this reason, persistent values are less useful for holding setup information. This role is usually filled by a disk file, or by using default values.
With the integration of Retained variables (see Retained Variables), the behavior of Persistent variables has changed as follows:

- Any data values supported by Pack may now be saved to disk (e.g. arrays, streams, link lists, etc.).
- The persistent size used for text string limits is no longer necessary. Any text string can be persisted, regardless as to its length.
- Recompiling a module does not delete the persistent variables.

Note: If you have an array (e.g. "A=New[10];"), and one element changes (e.g. "A[0] = 1;"), the modified value will not be known until the module has been stopped, as "A" itself did not change (its element did).

Retained Variables

Retained variables enable separate instances of a module to retain its value on disk between instantiations and VTScada executions. This is an enhancement on the existing persistent variables (see Persistent Variables). One application for Retained variables is for loading user settings; if the username changes, load all of the user's customized settings (i.e. set the instance name equal to username). In this way, you may use Retained variables like a database.

Note: With Retained variables, any parent instance name is inherited by all child modules. In the event that the parent's name is not defined, the application name will be used. Further, by default, all tags are launched with their name as the instance name, so you do not have to declare them.

The values for retained variables are stored in a special directory named, "Retained" that exists within your application directory. The files containing the retained variable values have the extension ".VAL".

Declaring Retained Variables

To declare a retained variable, you must use the "Retained" keyword before the variable definition. For example:

```
Retained My_Retained_Var { Retained Variable };
```

Assigning Names to Module Instances

Each module instance must be assigned a name. This can be done in one of the following ways:

- SetInstanceName() This function takes two parameters: Instance and Name. Instance is the object value of the module to which the name is to be assigned. Name is the text string name of the instance (please refer to SetInstanceName for detailed information on SetInstanceName()).

Note: With the inception of retained variables, all tags are launched with the tag name being the instance name of the tag.

- The name parameter in the Thread function also sets the name of an instance. (All applications are launched in a separate thread with the application name being the thread/instance name.)
- If an instance name is not explicitly set, an instance name is inherited from the nearest parent's name. If none of the parents have an explicit name, the name "Default" is used.

Setting the instance name will cause all the Retained variables to be reloaded with the values for that instance. Instance names may be changed dynamically.

Retained Variable Value Storage

Retained values are written to disk whenever they change. They are written using the Pack scheme, allowing complex arrays and linked lists to be persisted. Any modified values that are indirectly pointed to from the retained value will not trigger a rewrite of the retained variable to disk;

however, when the instance terminates, either through an instance name change or a stopping instance, the retained values will be written.
This implies that crashes, power failures, etc., can result in the most recent retained values not being on disk if these values contain pointers to values. As a result, retained values are stored in files with one file per instance, and one file per variable. These files are stored in a directory named, "Retained" within the same directory where the ultimate root module's .RUN file or the module containing the retained variable definition is stored. These files are uniquely named; the file name is made up of the names of parent modules concatenated with a dash (-) separator between module names, while the variable name is appended to this string by a plus symbol (+). The final '+' and instance name is not present if the variable is Shared. These files all contain the extension ".VAL".

Retained Variables and Statically Declared Arrays

Retained variables enable you to specify a default value for a statically declared array. If specified, all the elements of the array will be set to this value when the array is instantiated. For example, an array declaration such as:

```
[
   Data[10] = 0;
]
```

will result in all 10 elements of the data array to have the initial value of "0".
If the array with the initial value specified has a default value, the default value will only be used if there is no suitable retained value .VAL file. Recompiling to change the number or size of dimensions of a retained variable will cause any retained .VAL files previously saved with the old variable format to be ignored.

Note: Retained variables are similar in behavior to persistent variables (see Persistent Variables); in fact, persistent variables are equivalent to "Shared Retained".

Shared Variables

When running multiple copies of modules, each will keep a separate copy of its own variables. Prefixing a variable definition with the keyword "Shared" makes that variable a shared variable. Now when multiple copies of a module are running, and use this shared variable, all copies use the same value. For example, if one module writes the value 4 to a shared variable, X, all other copies of that module will see the value 4 in X. X is global to the module instances and only one memory location will be used for all instances of X. Shared variables are deleted only when the application stops running.

Saved Variables

Saved variables act like persistent variables without being shared. Prefixing a variable definition with the keyword "Saved" makes that variable a saved variable. Information on persistent variables can be found in Persistent Variables.

Network Values

The purpose of the Network Values service is to update changes on all PCs for remote applications when a change to a Network Value variable has occurred. Network Value variables cannot be used in a steady state.

Note: If your client is isolated from the server, and changes have occurred to Network Value variables, the client's changes will be overwritten when a reconnection to the server has been made.

Network Value variables are defined within the NetworkValue class, and are updated whenever the value for these variables changes on a client or server workstation. Network Value variables can store any value that you can Pack. (see note) Unlike Retained variables, Network Value variables recognize array element changes, and will update such changes as needed. It should be noted however that Network Value variables won't see multidimensional arrays; therefore, a subroutine called, "Update" can be called if you have changed a complex data structure like a

multidimensional array. During startup, only the Network Value variables whose value changed since the last synchronization are obtained.

Note: If using a multidimensional array with NetworkValues, you should note that the array must be dynamically allocated, rather than static. Also; you can store dictionaries, but you must use NetworkValues\Update to update them.

The values for Network Value variables are stored in a subdirectory named, "NetworkValues" within your application's directory. The name and format of these files is identical to that of the Retained variables (see Retained Variables); however, the extension for NetworkValues files is ".NV", whereas the extension for the Retained variable files is ".VAL".

Note: It is not recommended that you use the Network Values service for tag values, or other constantly changing values (e.g. mouse position), as the memory cost will be in the vicinity of .5 Kbytes to 1.5 Kb per Network Value variable. A better use for the Network Values service is for updates to other services.

Network Value Service Subroutines and Modules

In order to make use of the Network Values service, you must first start the service, and then call Register(). An example follows.

```
If \NetworkValues\Started Main;
[\NetworkValues\Register(Self, "MyModule");
   …
]
```

NetworkValues\Register

```
Register(Owner, InstanceName)
```

Where, Owner is the owner of the variables, and InstanceName is the name of the instance.

The purpose of the Register module is to allow any module with class NetworkValue variables to retain these variables across instantiations of a module and to have values automatically propagated around the network. If two instances of the same module register with the same

instance name, then each of these modules will be affected by the other's changes in values.
Register is a subroutine. After returning, the owner's variables will be set to the current server's version of the values. The caller must wait for NetworkValues\Started to become true before calling the Register module.

NetworkValues\Update

```
Update(Owner, VarName)
```

Where, Owner is the owner instance of the variables and VarName is the name of the variables, provided as a text string.
The Update subroutine should be called to force the value to be updated by the configuration server of a remote application, and to set the value in the NetworkValues file. This would be necessary if the value has changed in a way that cannot be automatically detected, such as by changing elements in an array that has more than one dimension or whose index does not start at "0".

NetworkValues\MonitorX

```
MonitorX(Owner, Ancestry, Head, Var1 [, Var2, Var3…])
```

Where...

- Owner is the owner instance of the variables.
- Ancestry is the list of parent modules, separated by a dash (-).
- Head is a pointer to the head of the list in the DBSystem.
- Var1 is the name of the variables (Var1, Var2, Var3, etc.)

Monitor modules are launched modules that watch variables to see if they change. If the value of the variables change, then the Monitor module checks to see if they differ from the server's version as represented in our database. If there is a difference, Monitor RPCs the updated value to the server. (The "X" implies that monitor modules can be named "Monitor1", "Monitor2", and so forth.)

NetworkValues\GetValue

```
GetValue(Ancestry, InstanceName, VarName)
```

Where...

- Ancestry is a list of parent modules, separated by a dash (-).
- InstanceName is the module instance name.
- VarName is the name of the variables.

The GetValue subroutine returns the value for a given variable within a given module specified with a specific instance name. The values are stored in a directory named, "NetworkValues" within your application directory. The name and format of the files is identical to that of the Retained variables (see Retained Variables); however, the extension for NetworkValues files is ".NV", whereas the extension for the Retained variable files is ".VAL".

NetworkValues\SetValue

```
SetValue(Ancestry, InstanceName, VarName, Value, Revision)
```

Where...

- Ancestry is a list of parent modules, separated by a dash (-).
- InstanceName is the name of the module instance.
- VarName is the name of the variables.
- Value is the new value to assign.
- Revision is the revision number.

The SetValue module sets the value for a given variable within a given module specified with a specific instance name. The values are stored in a directory called, "NetworkValues" within your application directory. The name and format of the files is identical to that of the Retained variables (see Retained Variables); however, the extension for NetworkValues files is ".NV", whereas the extension for the Retained variable files is ".VAL". SetValue is called via RPC.

NetworkValues\TestValue

```
TestValue(Owner, Ancestry, VarName)
```

Where...

- Owner is the owner instance of the variables.
- Ancestry is the list of parent modules, separated by a dash (-).
- VarName is the name of the variables.

You may use the TestValue subroutine to return true if the value passed in has a different value from the disk-based value.

Network Values Service Scheme

The Network Values service is designed to transfer values for any module around the system, and retain those values between starts for the application and instantiations of the module. The scheme used by the Network Values service follows.

1. DBSystem is created, containing Registered names concatenated with variable names, which are separated by a plus sign (+). The Registered names are also stored in the same DBSystem. Their value is the head of the list of Monitor modules. The values are saved in packed streams in files by the same name as the concatenated name. The retained registered names array also contains the version number for each value. The revision number is always incremented, and is a retained value.
2. RPC Synchronization is done, and Started is set.
3. The host module must wait for NetworkValues\Started.
4. Register(Obj, Name) is a subroutine that launches Monitor. Register performs a ListVars on Obj to get all class variables that will be distributed through remote procedure calls. It then launches sufficient Monitor modules to handle the number of variables. It also sets current values of variables from DBSystem before returning.
5. Monitor links itself onto a list wherein the registered name is the name of a variable in the RegisterNamesDB containing the object value of the head of the list. The list is self-repairing, and not ordered.
6. Monitor watches for changes in simple values and 1 dimensional array of values for the variables and then checks if the current value is different from the DBSystem value. If different, an RPC message to the server is sent to set the new value.
7. The server sends the SetValues message to all clients and itself.
8. The persistent values are stored in a directory called NetworkValues. Each value has its own file. Each file is a member of a linked list whose head is NetworkHead, and is sorted by the Revision number. This enables the GetServerChanges to easily get the most recent values.

9. SetValues sets the value in the DBSystem, writes the value to the corresponding file for the name, walks through the list of Monitor modules for the registered name, and sets the variable in each of the registering modules.
10. If a registering module stops, the Monitor automatically stops since it has been called from the registering module.

Note: The RAM used for an actively monitored value ranges from about 550 bytes when 8 or more variables are in the module, to about 1500 when only one variable is in the module.

Temporary Variables

In cases where an application is being modified online, it may be useful to create short-lived variables that will disappear when VTScada is shut-down and restarted. These are known as temporary variables. Temporary variables are kept in RAM while VTScada is running, so they will continue to exist even though the application that uses them is stopped and restarted. They are not written to the .RUN files of the application, though, so once VTScada is stopped or the application is recompiled, they will cease to exist. For the application to use them again, it must once more create them using an AddVariable statement.
Temporary variables should not be referenced like other variables, but should be accessed using one of the following functions to reduce potential problems in their use:

- FindVariable
- ListVars
- Scope
- VarAttributes
- Variable

Protected Variables

On occasion, it may be necessary to limit access to certain variables to those objects lying within the variables' scope. This is done by prefacing the variable declaration with the keyword "Protected".

Protected variables (and modules) are not accessible through scope resolution.

Variable Class Definitions

A variable may be of a defined class, if so designated by the user. Variable classes are primarily used as a means of grouping associated variables together and may be defined as the user requires them. To set a variable's class, the SetVariableClass function may be used, or the class number may be added to the variable's declaration or to the declaration of a group of variables of the same class:

```
[
   FlowRate = 32 (0x0015);
   PNum (0x0015);
   X;
   Y;
]
```

In this example, the first two variables have their class designated by the hexadecimal number 15 (0x0015), although in the case of FlowRate, a default value of 32 has also been set. Since these two variables have the same class designation, they could have been grouped together as follows:

```
[
  Constant PUMP 0x0015;
   [ (PUMP)
     FlowRate = 32;
     PNum;
  ]
 X;
 Y;
]
```

Notice that in this case, the variable class for FlowRate and PNum is a constant variable called PUMP whose value has been set at 0x0015 in a previous declaration statement. No matter which method is used, the class definition must be a constant or an expression that evaluates to a constant, whose value is between 0 and 65 535.

Example:

Each tag type will have a class. This can be used to obtain a list of the tags of a certain type. For example:

```
IF ! Valid(MyClass); [
   { Get the Alarm Tag's class }
   MyClass = VariableClass(FindVariable("AlarmPriority", \Code, 0,
0));
   { Find the array of variables that are members of that class }
   PriorityObjs = ListVars(ParentObject(\Code), "*", MyClass,
MyClass, 0, 0, 1, 0, 0);
]
```

Related Information:

...Variable Storage, Retention, Access

...VTScada Value Types - Numeric Reference

VTScada Value Types - Numeric Reference

The following table lists the value types used in VTScada. When referring to these in code, you should use the predefined constants rather than the type numbers. The general usage is:
Cast(Val, \#VtypeText)

Type	Constant Name	Name	Description
0	#VTypeStatus	Boolean	Logical data type, stores two states: "true" (0) or "false" (non-zero).
1	#VTypeShort	Short, 16-bit signed	Integer data type storing values from -32768 to 32767
2	#VTypeLong	Long, 32-bit signed	Integer data type storing values from -2147483648 to 2147483647
3	#VTypeDouble	Double precision floating point	Values range from about -10^308 through +10^308
4	#VTypeText	Text	Any string of bytes whose values range from 0 to 255. Typically used to hold text strings.

5	#VTypeVariable	Variable	A handle to the data represented by a variable declaration, not to any particular instantiation of that declaration. Can be used to access variable metadata (type information, for example) or default values.
6	#VTypeFunction	Function	A pointer to the code for a particular function within a VTScada statement. Used by functions such as GetOneParmText to manipulate the code itself. Used when compiling and editing script code, not for typical VTScada programs.
7	#VTypeObject	Object value	An instance of a module
8	#VTypeStream	Stream	A handle to a stream (of which there are several types). See Streams.
9	#VTypeModTree	Module tree	A handle to the modules in a state diagram
10	#VTypeStateDgrm	State diagram	A graphical depiction of VTScada code
11	#VTypeModule	Module	The code and variables that make up a unit of a VTScada program. See Modules.
12	#VTypeModState	Code Value (a) Module and state	A handle to a state within a module. See States.
13	#VTypeModStateStmnt	Code Value (b) Module, state, and statement	A handle to a statement within a state. Cannot refer to any arbitrary function, as type 6 can. See Statements and Graphic Objects.

14	#VTypeRefParm	Reference parameter	When a steady-state call is made to a module, each of the actual parameters in the call is "bound" to its corresponding formal parameter.
15	<undefined>	Array	Refers to an entire list of consecutive data values. Each data value has a consecutively numbered index address and may be any VTScada value. See Array Variables
16	#VTypePath	Path	A series of vertex values. See Path Variables.
17	#VTypeTraj	Trajectory	A combination of a Normalize value and a Path value. See Trajectory Variables.
18	#VTypeRotate	Rotate	Specifies a rotation amount, measured in degrees, around a point. See Rotate Variables
19	#VTypeBrush	Brush	Brush values are used in layered graphics statements that paint areas of the screen with a uniform color or pattern. See the Brush function.
20	#VTypePen	Pen	Pen values are used in layered graphics statements that draw lines. Defines the color, style and thickness of a line. See the Pen function.
21	#VTypeNormalize	Normalize	A graphical scaling value. See Normalize.
22	#VTypePoint	Point	A location, stored as an (X, Y) pair. See Point.

23	#VTypeVertex	Vertex	A group of three Point values. See Vertex.
24	#VTypeTransform	Transform	A transformation matrix, used to map coordinates from one area of the screen to another. Can only be obtained from the GetTransform function. Used by the GetPathBound function.
25	#VTypeCodePtr	Code pointer	A handle to an active graphics statement in a particular module or state. Similar to type 13, but with the additional information of the module instance as represented by value type 7.
26	#VTypePtr	Pointer	Stores data by reference instead of by value, allowing, for example, multiple values to reference the same piece of data as opposed to multiple copies of the data.
27	#VTypeEditor	Editor	A handle to an editor object, as created by MakeEditor.
28	#VTypeParseStack	Parser stack	Used by the compiler to allow the compilation to be suspended in the middle of a statement to handle specific code sections such as I/O addresses.
29	#VTypeTag	Tag	(Unused) Intended to provide engine-level support for scaled variables that could be implemented using a GUI.
30	#VTypeBitmap	Bitmap	A handle to an image object as returned from MakeBitmap.

31	#VTypeFont	Font	A handle to a font object, as returned by the Font function.
32	#VTypeVTSdb	VTScada database	A handle to the VTScada database as returned by the DBSystem function.
33	#VTypeODBCHndl	ODBC Handle	Provides a connection to an ODBC database.
34	#VTypeSAPIStrm	SAPI text-to-speech stream	A type of stream for use with Speech Application Programming Interfaces
35	#VTypeComClient	COM Client Interface	An object that provides an interface to a COM client application
36	#VTypeCryptoProv	Cryptographic Provider	A handle to the particular cryptographic service provider that includes the key specification to use.
37	#VTypeCryptoKey	Cryptographic Key	May be either a Session Keys or a Public/Private Key. See Cryptographic Keys.
38	#VTypeDLLhandle	DLL Handle	A pointer to a structure returned from the LoadDLL function. Used to call functions within the DLL that was loaded. See DLL.
39	#VTypeDeflateHandle	ZLib Compression Handle	Used by the Deflate function
40	#VTypeThread	Thread Handle	A script-level hook to the data structure used to represent a thread in a dump
41	#VTypeBreakWatch	Source Debugger Breakpoint Handle	References a set location in the source debugger. See Working with Breakpoints and Data Break-

			points
42	#VTypeMiniDumpHandle	Minidump Data Handle	A pointer to a data structure that holds information from a crash dump
43	#VTypeTimeStamp	Timestamp	A numeric representation of time, measured in seconds since January 1, 1970
44	#VTypeXMLproc	XML Processor Handle	Serves as a conduit between an XML document and an application. See VTScada Engine XML API
45	#VTypeTypeDefinition	Dynamic Module Definition	Deprecated. A handle to the definition of a form of module used as a data container. Created by the MakeType function. This storage is used almost exclusively for handling XML and cannot contain script code (unlike other forms of Module).
46	#VTypeTypeInstance	Dynamic Module Instance	Deprecated. An instance of a dynamic module, created using the MakeTypeInstance function. This is an object value (type 7) that can only be used to store data - it cannot contain or execute script. Typically these are used when generating module trees for delivery via XML. It is a form of data container, however in general structures (defined by the Struct function) and Dictionaries (type 47) are more efficient and convenient for this role.

47	#VTypeDictionary	Dictionary	A key-based data container of flexible size, used either on its own to hold volatile data collections or in the definition of structures (see Structures). ValueType will not return this value unless the dictionary is a "pure" dictionary. A pure dictionary is one for which the root value has not been set. Otherwise, it returns the ValueType of the dictionary's root instead. See Dictionaries
48	#VTypeComProperty	COM Property	A value exposed by a COM Interface "object". This may be accessed similarly to a typical VTScada value but is maintained by the COM object, not the VTScada engine.
49	<undefined>	Module in Context	Contains both a module value and an instance of the context module where scope should be resolved. Normally, scope will be the parent module in which the Module was declared. A Module in Context is used for widgets and plug-ins in VTScada where the widget is declared in AppRoot.SRC, but linked into a tag type such that the widget becomes a Module in Context in the tag instance. References to variables in the widget will then

			refer to variables in the tag rather than to variables in AppRoot where the widget was declared. If a Module In Context value is called in steady-state, the parent instance will provide the associated context.
50	#VTypeHistorianHandle	Historian Connection Handle	For the VTScada proprietary data store, this will be invalidated on an "out of disk space" error, or on loss of access to the file storage. For other databases, this will be invalidated on any connection loss.
51	#VTypeXMLNode	Dictionary Structure	A WEB_XML_ADDRESS that points to a WEB_XML_NODE. When ValueType() runs against a value and finds a WEB_XML_ADDRESS it treats it the same as a WEB_VALUE_ADDRESS, which sits in front of an array or structure. It then searches through the *_ADDRESS to find what it points to and returns the type of that item, in this case an USER_XML_NODE
52	#VTypePPPHandle	PPP Connection Handle.	May be passed into the function, PPPStatus() to obtain an information structure. May be passed to the function, CloseStream() to forcibly close off a connection. Passing it into CloseStream completely inval-

idates the handle and all data associated with it.(see: PPPStatus and CloseStream)

Style Guide for VTScada Code

Developers at Trihedral use this style guide in order to create consistent and easy-to-read VTScada code. If you follow the recommendations in this guide it easier for others to read your code and for you to read others' code.

Comments

1. Start the VTScada file with a multi-line, 80 characters-per-line comment describing the purpose of the module. The first line of the comment should contain the name and ancestry of the module, centered between lines of equal signs:

```
{====== PageManager\PageViewer ======}
```

   Note that the number of equal signs varies - whatever is required to pad out the comment to 80 characters. The last line of the comment should be strictly equal signs, used to mark the end of the comment. A single space is left between the equal signs and the text on either side.

2. Full line comments (on their own line) that are used to describe the action of statements, etc. should have 5 asterisks after the opening curly brace and 5 before the closing curly brace of the comment:

```
{***** This statement blah blah blah *****}
```

3. Comments generally begin with a capital letter, however, single or few word comments may or may not use capitalization, as appropriate.
4. Multi-line comments should use a single set of braces; if the comment is a full line comment and will have asterisks, the text should be left aligned (i.e. indented the width of the asterisks).
5. All variable and parameter comments should start in the same column (preferably column 32) and end at the same column. This type of comment

is not a full line comment, so it should not use the asterisks described in item 2.

6. The closing angled brackets for a module should be preceded by a comment indicating the end of the module. Although it is on its own line, it should not have the asterisks mentioned in item 2:

```
...
{ End of PageManager\PageViewer }
 >
```

7. Any code which is inserted for testing or for other temporary purposes must have a comment which includes two consecutive question marks { ?? } which can be rapidly located before code is shipped.

Variables and Parameters

1. Start parameters with an opening bracket in the leftmost column on the line immediately following the module name.
2. Indent all variable and parameter definitions two spaces.
3. All variables and parameters must have some comment.
4. Use descriptive names for variables and parameters.
5. Variable and parameter names should begin with a capital letter.
6. Place the semi-colon for the variable or parameter declaration after the comment on the same line - this makes it possible for the VTS application manager to properly associate the comment with the correct variable.
7. Do not exceed column 132 so that it can be easily viewed in a DOS editor and printed without loss of visible characters or wrapping.
8. For default values, align all = in the same column.
9. Blank lines may be placed between variable groups that have different purposes. A comment identifying the purpose of the group should be provided.
10. Constants may be declared with all upper case names, as appropriate.
11. Constants used for indices should begin with the # character.

```
{============================== CustomTag
================================}
{ Custom tag definition. }
{=================================================================-
=======}
(
  Name <:TagField("SQL_VARCHAR(64)", "Name",       0 ):> { Name of
this tag };
```

Modules

1. With module declarations, align all "Module" keywords in the same column.
2. Module declarations with file names should align the file names in the same column.

States

1. State declarations should be preceded by a blank line.
2. State name declarations should be in the first column.
3. State names should begin with a capital letter.
4. The square bracket beginning the state declaration should be on the same line as the state name, separated from the name by one space.

Statements, Functions and Operators

1. Statements in a state should be indented by two spaces.
2. No spaces should be on either side of the brackets for a function.
3. All commas should be followed by a space
4. Assignment statements which do not consist of functions with long parameter lists which can be split on multiple lines can be split so that the text in subsequent lines lines up with the first character after the =.
5. Where multiple = statements are listed together, align the = in the same column.
6. Place a space before and after all two-parameter operators such as + and =, with the possible exception of those found inside the square brackets of an index of an array element.
7. The prefix operators * and & should not have a space after them; the prefix operator ! should have a space between it and its variable.
8. Semicolons terminating a statement or declaration should follow immediately at the end of the line without any preceding spaces.
9. Indent script statements a total of four spaces.
10. Write all functions with an upper case letter. Use a mixture of upper and lower case where appropriate (CamelCase)

11. All functions should have a set of brackets, regardless of whether or not they require parameters; this makes it easy to differentiate between a function and a variable.

WhileLoop
DoLoop
IfElse

1. Place the conditions for the WhileLoop statement on the first line with the body of the loop indented two spaces from the WhileLoop itself on the following lines. Terminate the WhileLoop with the) on a separate line aligning it in the same column as the WhileLoop.
2. IfThen statements have their condition on the first line with the body of the code and the termination the same as for the WhileLoop.
3. Start the DoLoop statement with no parameters on the first line. Indent the body of the loop two spaces from the DoLoop. Terminate the loop with the condition at the same column position as the DoLoop followed immediately by the) on the same line without any spaces.
4. IfElse has its condition on the same line as the IfElse. If an Execute is required for the TRUE case, it is placed on the first line as well, with its opening bracket immediately following it (no space) and its closing bracket on its own line in the same column as the IfElse. The "else" case is started with a comment { Else }in the same column as the IfElse. If it requires and Execute, the same guidelines are followed as described previously, except that the closing brackets for the Execute and the IfElse are kept together on the last line with no space in between.

```
IfElse (Var1, Execute(
 ...
);
{ Else } Execute(
 ...
));
```

Example

(Comments do not extend the full 80 character width here due to space limitations. This is not functional code.)

```
{=================== CustomManager\RemoteConfig
=====================}
{ Dialog displayed when ( ) button pressed
}
{=========================================================================-
=}
(
  Trigger                       { Flag - TRUE when dlg is displayed
};
)

[
  Dialog Module "RBDlg.SRC"        { The dialog
window                 };
  DefaultThings Module "Def.SRC"   { An alternate dialog
window       };

  Persistent XPos         { The x-position of dialog
window             };
  Persistent YPos         { The y-position of dialog
window             };
  Constant HT = 532       { The height of the dialog
box                };
  Constant WD = 480       { The width of the dialog
box                 };
  InitVar                 { Initial value of var to be
changed          };
  MaxChars                { Length in chars of longest
label            };
  Reset = 0               { Flag - TRUE to reset
vars                   };
  ShowWindow = 0          { Flag - TRUE while window is
open            };
  Title                   { Title of the window
};
  GIZMO                   { Pointer to the GIZMO module
};
]

Init [
  If 1 Wait;
  [
    {***** Find the length of the longest label *****}
    MaxChars = Max(StrLen(\FileLabel), StrLen(\PageLabel),
                   StrLen(\TagLabel)) + 1;

    {***** Define the title based on the user settings *****}
    IfElse (UseName, Execute(
      GIZMO = Scope(Caller(Self()), "------");
      Title = Concat(GIZMO\App, " - ", \RemoteConfigLabel);
    );
    { Else }
      Title = \RemoteConfigLabel;
    );
```

```
    {***** Perform initialization tasks *****}
     DefaultThings(&InitVar);
  ]
]

Wait [
 {***** Open the dialog when the user requests it *****}
  If Trigger OpenDlg;
  [
    ShowWindow = 1;
    Reset = 0;
  ]
]

OpenDlg [
  {***** The specified button was pressed or another dialog was
opened via the toolbar *****}
  If ! Trigger Wait;
  [
    ShowWindow = 0;
  ]
  If ! ShowWindow Wait;
  [
    ResetParm(Self(), 1);
  ]

  Window(PickValid(XPos, 100), PickValid(YPos, 100),
         WD, HT, WD, HT,
         Dialog(), \DialogWin,
         Title, \DialogBGColor, ShowWindow);
]
{ End of CustomManager\RemoteConfig }
```

Basic Programming Tasks

While the range of what you can achieve with the VTS language is best described by the word "anything," most projects will include several fundamental components such as opening a window and displaying graphic elements, or gathering user input. Additionally, many people learn VTS coding for similar ends, such as building a new type of report or creating a new type of tag.
This chapter is provided as a guide to these common and basic tasks. It begins with instructions for creating a new script application, or adding a new module to a VTScada application.

Related Information:

...Create a New Script Application

...Add a Module to a VTScada Application

...Working with Pages

...Create Windows & Use Graphics Functions

...Obtaining User Input

...Time and Date

...Build Custom Reports

...Working with Speech

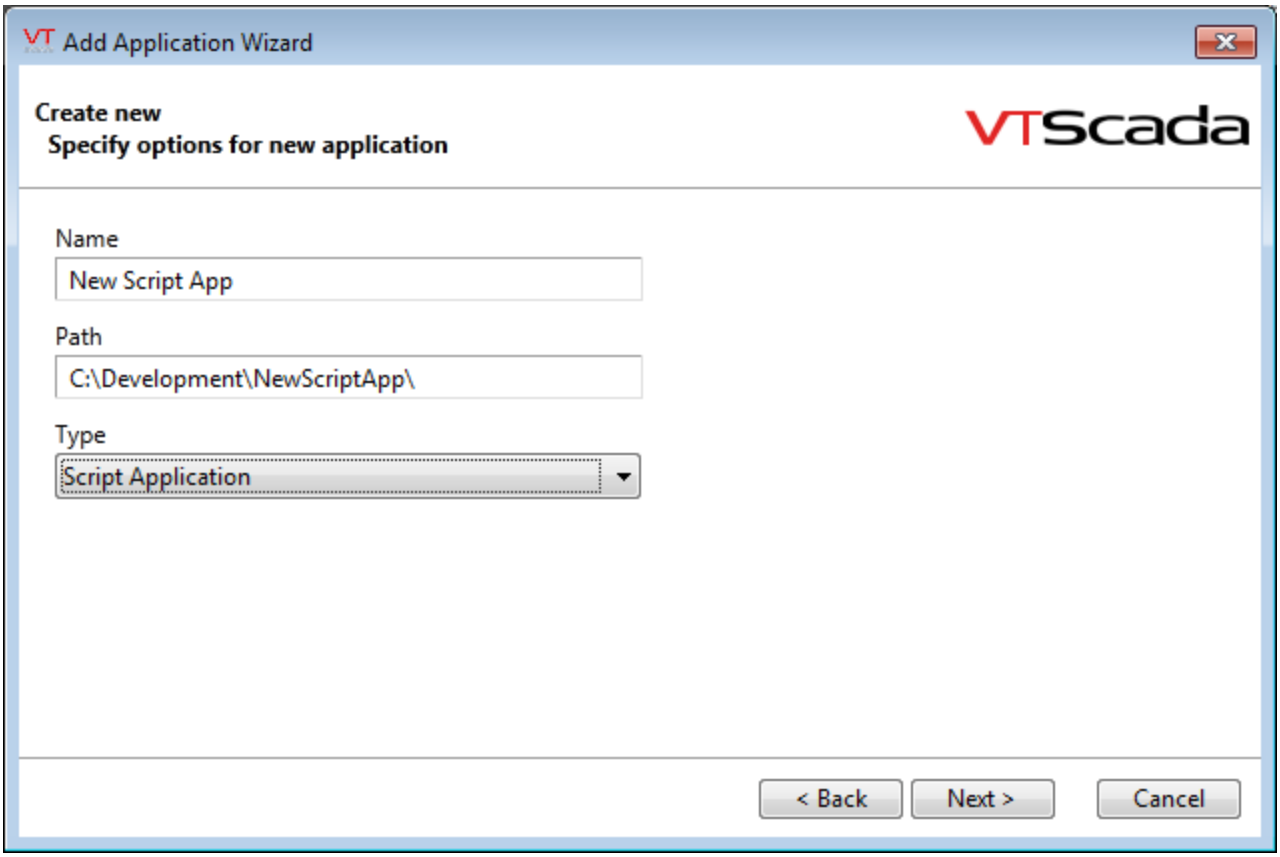
...Interrupt the Shutdown Process

Create a New Script Application

A VTScada script application is one that is not based on the VTScada layer, or on any other OEM layer, and therefore does not provide access to the common VTScada development tools and services such as the Display Manager, Idea Studio, Alarm page, etc. A script application must be programmed from start to finish.

Script applications are typically created for the purpose of analyzing data or as utilities to perform custom tasks (such as converting databases).
To create a new script application:

1. Click the Add Application Wizard button (plus sign) that appears in the VTScada Application Manager (VAM).
   The wizard dialog opens.
2. Select the Advanced radio button, then click Next.
3. Select the Create new option, then click Next.
4. Enter a name for the application in the Name field and press Enter or Tab.
   The Path field is automatically filled with the name of the application directory for this new application based on the application name you've entered minus the spaces. You have the option of changing the path if you would like.
4. Choose Script Application from the Types drop-down list.



5. Click, Next.
6. Verify the options for the new application and click Finish.

VTScada generates the new script application, after which it will be listed in the VAM.

A new application folder is added to your VTScada installation directory. Within this new application directory, you will find the file, AppRoot.src. This is the main application module file for your VTScada application, listing all top-level variables and modules for the application.

For most script applications, AppRoot will have a single state containing a single statement: a call to the Window() function, passing in the name of a submodule (Graphics), which will run in the context of that window. In most cases, you will add your code to the module, Graphics, or to submodules of Graphics that you create.

Given the script application created using the preceding steps, you can create a simple "Hello World" application as follows:

1. Using a text editor (Notepad or similar) open the file AppRoot.SRC in the new application's directory.
2. Edit it to match the following example.

```
{================================== System
==================================}
{=============================================================================-
=========}
(
  System                        { Provides access to system library
functions };
  Layer                         { Provides access to the application
layer    };
)
[
  Graphics               Module { Contains user graphics
};
  WinTitle = "Greetings!" { Window title
};
]
Main [
  Window(  0,   0           { Upper left corner   },
         800, 600           { View area           },
         800, 600           { Virtual area        },
         Graphics()         { Start user graphics },
          {65432109876543210}
         0b00010000000110011, WinTitle, 0, 1);
]
<
{============================ System\Graphics
==============================}
{ This module handles all of the graphics for the application
}
```

```
{=========================================================================-
=========}
Graphics
Main [
  ZText(100, 100 { Lower left corner of text },
         "Hello World" { Text to display },
          15 { Text is white },
         0 { Use default font });
]
{ End of System\Graphics }
>
```

3. In the VAM, click the Import File Changes button for the application.
4. Provide a comment and click OK.
5. Run the application.

Troubleshooting:

AppRoot.SRC looks nothing like the example here. There is no Graphics submodule

- The application was not created as type, Script Application. You will need to go back to the VAM and create a new application.

The window is blank.

- There are three possible reasons: 1) The coordinates given to ZText are outside the window area. 2) The text color provided matches the background rather than being white (15) on black. 3) You skipped step 3 – import file changes.

An error message is displayed when you click Import File Changes.

- There is a typographic error in your code. Check for spelling, commas, semi-colons after every statement and a closing quotation mark and bracket for every opening one. The message will give you a starting point to look for the error, but you may need to scan up or down in the file to find the actual source of the trouble.

Example:

The Bonus Program

"Bonus" - a reward, usually financial, distributed to employees at the end of a year.
The bonus program is a script application that is used in every VTS programming course, to teach the basic concepts of the language. Longer than a "Hello World!" program, it shows the workings of modules, states, showing how to change graphics pages and how to interact with an operator by monitoring the position of the mouse pointer. By studying this example, line by line, you can learn a great deal about how VTScada programs work. For example, why do the contents of screen 1 vanish when screen 2 only contains code to add graphic elements, not remove any? (* answer at bottom of page.)
The code for the bonus application is provided here for you to use in a new application of your own. Once you have the application working, you may wish to enhance it by adding new pages, graphic elements or user-input elements.

1. Create a new script application, and name it "Bonus".
   Follow the steps in the preceding topic, Create a New Script Application.
2. Using a text editor, open the file "AppRoot.SRC " found in your new application's folder.
3. Replace the contents of AppRoot.SRC with the code following these steps.
4. Save the file. (It is recommended that you leave the editor open, so that you can easily fix errors, or move on to experiment with new code.)
5. In the VAM, click the application's Import File Changes button. Provide a comment when prompted.
6. Run the application. You can reset it to the first screen by pressing the "Esc" key.

```
{================================== Bonus
====================================}
{=========================================================================-
========}
[ { Primary module }
   Graphics Module { Sub-module declaration. Contains user graphics
```

```
};
   WinTitle = "The Bonus Program" { Window title };
   System { Provides access to system library functions};
]
Main [ { the only state contained in this module }
   { function call to create a window }
   Window( 0, 0 { Upper left corner },
           800, 600 { View area },
           800, 600 { Virtual area },
           Graphics() { Start user graphics submodule, which provides
the window contents},
           {65432109876543210}
          0b00010000000110011, WinTitle, 0, 1); { bit-wise control
the window appearance }
] { end of the state code }

< { Sub-module }
{============================= System\Graphics
==============================}
{ This module handles all of the graphics for the application }
{=========================================================================-
========}
Graphics { module name }
(
 { Parameter section. Empty in this example. }
)
[ { Local variable declarations }
   X = 360  { X Position of Button };
   Y = 280   { Y Position of Button };
   FontValue { Variable for Font Value };
]
 { first of two states in this submodule }
Screen1 [
   ZText(230, 150, "Click here for an Important Announcement", 14,
0);
   If ZButton(340, 175, 420, 225, "Notice", 1) Screen2;
]
 { second of two states in this submodule }
Screen2 [
   FontValue = Font("ARIAL", 0, 30, 0, 5, 0, 0);
   ZText(150, 150, "Well, It's Bonus Time Again!", 14, FontValue);
   ZText(270, 230, "Click here for this Year's Bonus", 14, 0);
   ZBox(360, 280, 440, 330, 224);
   ZButton(X, Y, X + 80, Y + 50, "Bonus", 1);
   If Target(X, Y, X + 80, Y + 50);
   [
      X = Cond(XLoc() < X + 40, X - 120, X + 120);
      Y = Cond(YLoc() < Y + 25, Y - 90, Y + 90 );
   ]
   If MatchKeys(1, Makebuff(1, 0x1B {Esc})) Screen1;
   [ { reset the x and y positions }
      X = 360 { X Position of Button };
      Y = 280 { Y Position of Button };
   ]
]
{ End of System\Graphics }
>
```

Troubleshooting:

- The application won't compile.
  There is a typographic error in your code. Note the line number given in the error dialog. This gives you a starting point for locating the error.

(*) Screen 1 vanishes because, upon leaving a state, all the code that was running in that state stops running.

Related Functions:

... Cond

... Font

... MatchKeys

... Target

... Window

... ZBox

... ZButton

... ZText

Add a Module to a VTScada Application

In smaller script-based applications, you will write your code directly in the AppRoot.SRC file, as shown in the bonus application example. (The Bonus Program)
This will not be the case with a VTScada application. There, AppRoot.SRC will contain only declarations of constants and submodules; not module code. If your goal is to enhance a VTScada application by adding a new tag, driver, report, data-entry wizard, etc. then follow these steps:

1. Using a text editor, create a new file in the application folder.
   The new file should be given a name that matches what you intend to call your module, and must have the extension, ".SRC".
2. Write your VTS code in that new file.
3. Using a text editor, open AppRoot.SRC in the application's folder.

4. Add a line to declare your new module and give it a name.
   This line must be in a section of AppRoot.SRC that is appropriate for the type of module you are adding. For example, new tags must go into the (POINTS) section. New reports are declared in the (PLUGINS) section.

Examples:

...A 15-Minute Snapshot Report

...Hide the VAM from Operators, but not Managers

Related Information:

...Working with Pages - code can be changed in or added to pages and user widgets within an application

A 15-Minute Snapshot Report

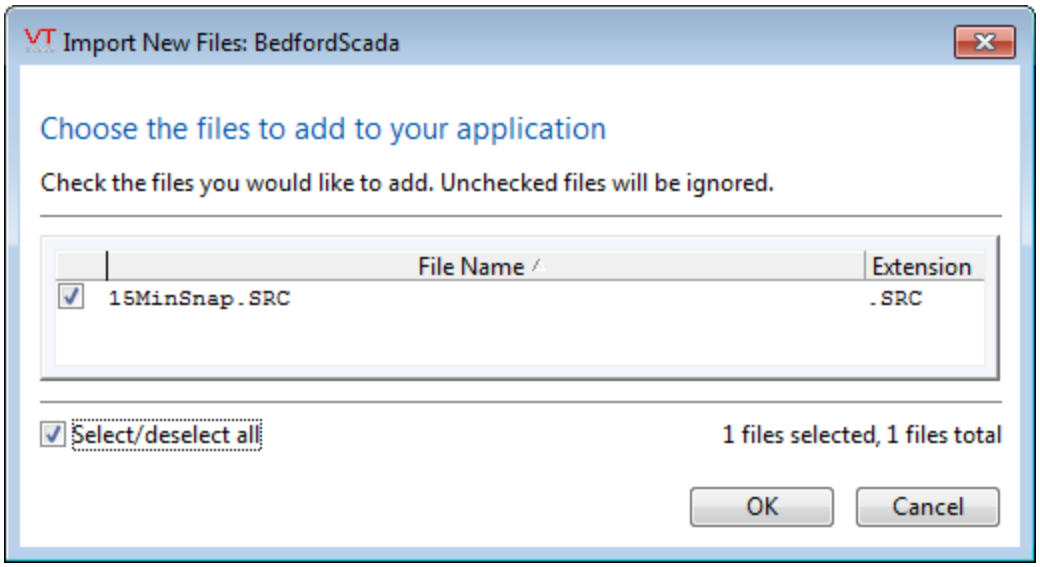
This example shows how to create a new type of report, and how to add a new module to an existing VTS program. The result will be a snapshot report that works on a fifteen-minute basis rather than hourly or daily.

1. Select an existing application, or create a new one.
   Do not select or create a script application.
   Do not risk disaster by experimenting within a running production application.
2. Using a text editor, create a new file in that application's folder.
3. Name the file "15MinSnap.SRC".
4. Copy the code following step 10 into that file and save it.
5. Using a text editor, open the application's AppRoot.SRC file.
6. Declare the module within the (PLUGINS) section.
   The result should appear as follows. Note that the filename is case sensitive - you must enter upper and lower case letters in the declaration, exactly as you named the file.

```
[ (PLUGINS) {===== Modules added to other base system modules =====}
   15MinSnap Module "15MinSnap.SRC";
]
```

   (There will already be a (PLUGINS) section - do not add a second one.

7. Save the file and click the application's Import File Changes button.



8. Click OK to import the new module.
9. Start the application if it is not already running. (It was not necessary to stop it to do the preceding steps.)
10. Open the Reports page. Your new report should be available in the list of report types.

```
{================================ 15MinReport
==============================}
{ This plugin modifies the hourly snapshot report to be every 15
minutes      }
{ Groups : Loggers
        }
{ Areas : All
        }

{=============================================================================-
=========}
(
   Reporter { Object value for call-backs };
   Start { Starting time };
   End { Ending time };
   Tags { List of tag names to report on };
   Vars { List of vars within tags };
)
[
  { Set up this module to become a plug-in for the reports }
   [(POINTS)
      Shared Report;
   ]

   Constant TypeFilter = "Loggers" {type of tags to use in the
report};
   Constant ReportName = "15 Minute Snap" {title for the report };
   TimeStamp { Time of last value returned };
```

```
   Obj { Instance of report };
]

Init [
   If 1 Wait;
   [                                                { 15 minutes = 900
seconds }
      Obj = \SnapshotReport(Reporter, Start, End, Tags, Vars, 900,
ReportName, 4 );
   ]
]
Wait [
   TimeStamp = Obj\TimeStamp; {ensures that the report object was cre-
ated before this module ends }
   If !Valid(Obj);
   [
      Slay(Self, 0);
   ]
]
```

Troubleshooting:

- The application won't compile.
  There is a typographic error in your code. Note the line number given in the error dialog. This gives you a starting point for locating the error.
- The report is not available.
  Ensure that you typed the code exactly as shown.
  Ensure that the declaration was placed in the existing (PLUGINS) section of AppRoot.SRC, and was placed before the closing square bracket of that section.
  Ensure that the Load File Changes button was pressed and no error dialogs opened as a result.

Related Information:

...Build Custom Reports - Discussion and instructions for creating custom reports

Hide the VAM from Operators, but not Managers

The HideVAM application property can be set true to hide the VTScada Application Manager (VAM) from view. While useful, this is not especially convenient since it depends on having an application set to auto-start, and at least one user account in that application granted the privilege of seeing the VAM.

You can create a service module for any application that, while the application is running, will control the value of HideVAM on a workstation based on privilege or other property of a logged-in user.
The complete text of the module is as follows. It uses the SecurityCheck function to inquire as to whether the logged-in user possesses the Manager privilege (PrivBitManager). HideVAM is then set to the opposite of the test for this privilege (thus the VAM is not hidden if the privilege is set).

Note: You must use HideVAM rather than the older version, HideWAM. HideWAM is now checked only on startup, and is only used to set the initial value of HideVAM.

```
{========================= HideVAM ==============================}
{ Hides the VTScada Application Manager based on security level   }
{=================================================================}
[
  CanConfigure             { Set if we can view VAM               };
]

Run [

  CanConfigure = PickValid(\SecurityManager\SecurityCheck(
                       \SecurityManager\PrivBitManager, 1), 1);

  If Watch(1, CanConfigure);
  [
     { Set the WAM's visibility based on security manager check }
    \SysLib\HideVAM = !CanConfigure;
  ]
]
{ End of HideVAM }
```

You may choose any security privilege you wish – see: System Privilege Reference for the complete list.
The module should be declared in the [Services] section of the application's AppRoot.SRC file.

Related Information:

See: "Hide the VAM" in the VTScada Developer's Guide

Working with Pages

You can change the appearance and behavior of the pages in your application. There are two levels of access to the characteristics of pages: at the developer's level, page characteristics are set using the Idea Studio and its associated tools. At the programmer's level, page characteristics can be set within a page's source file. Developer tools are described elsewhere in this guide.

Note: The code for user-created widgets is very similar to that of page modules, and therefore much the following information also applies to them.

Caution: In general, it is not recommended that custom code be layered on top of existing page object code. For example, by attempting to make a Page Close button perform extra tasks before executing its own page-closing code. This may easily result in a race condition. Create dedicated code for each task instead.

Where to find the code for your pages:
For each page, there are three resource files,stored within your application in a folder named, "Pages". These are user-editable copies - any change must be imported into the application by an authorized user before it will become part of the running application.

- PageName.SRC The .SRC file contains the source code for the page.
- PageName.RUN The .RUN file is the last compiled version of the page.
- PageName.BAK The .BAK file is a backup of the source code for the page.

The code of a page is a standard VTScada **module**[1], following all the rules of the VTScada language. It may begin with an optional set of parameters

```
(
   parmName <:"description text":> data_type;
)
```

Following the parameters (if any) will be the variable declarations.

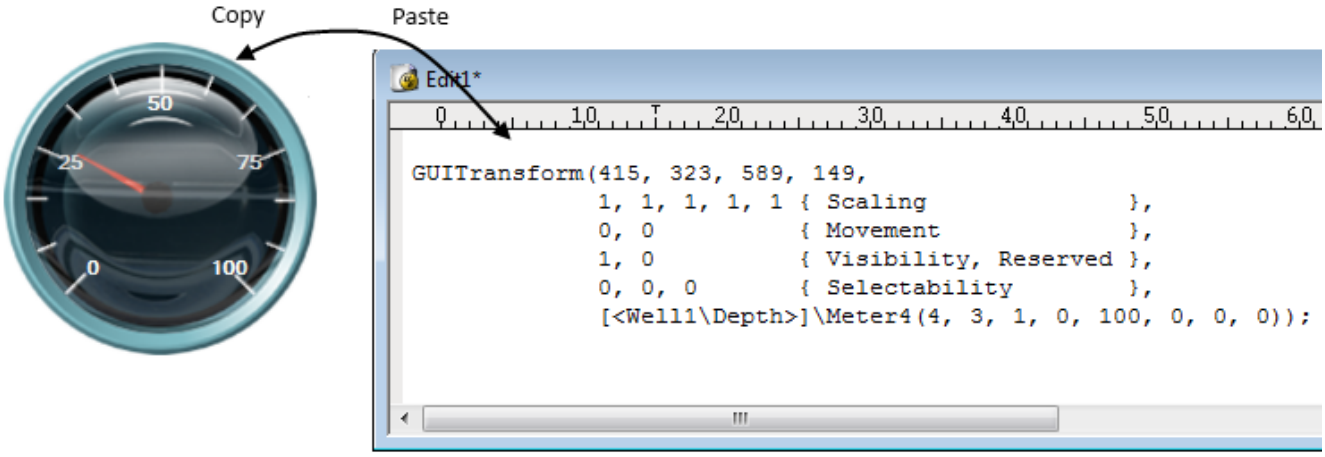
```
[
   Title = "Overview";
   Color = 89;
   ...
]
```

Most pages will have one state, named "Main" by default when the page is created within the VTS development tools. This is a subroutine, and therefore will contain the statement, Return(Self);
All graphic entities added to a page within the development environment will use the GUI- functions (GUIPipe, GUITransform, etc.), but you are free to use Z- functions if you are adding objects using code.
The order of the graphical statements within the page's state determines their display order (later statements have a higher z-order than earlier statements) and also the tab-order for user-input elements.
Tags are drawn using drawing-method code, placed within GUITransforms. The drawing-method code controls the appearance (meter, top-bar, slider, etc.). The GUITransform controls the scaling and location.
Within the page code, each tag is referred to by its unique ID. This enables developers to rename tags at will without losing the associated graphic. If you copy the graphic, then paste it into an editor, VTS will automatically look up the current name of the tag, and use that instead.

---

[1] A collection of states, scripts, variables, parameters, comments and possibly other modules, all of which make up a VTS program. Modules are separate tasks that run simultaneously in an application. Behind every page is a module and behind every tag on that page is an instance of another module (usually with submodules controlling separate tasks).

(The absolute name will be used, rather than the relative name, as indicated by the angle brackets.)



Variables within a Page's Source File

All of the page elements that are accessible from the page editing dialogs are stored in the page's source code file and can be modified there.

DrawLabel

The DrawLabel variable enables you to specify a title for the page. When used within a page's source code, the DrawLabel variable identifies not the text to use, but the name of a configuration variable whose value specifies the text to be used as the page's title. For example, when DrawLabel is declared in the page's source code file as follows:

```
CONSTANT DrawLabel = "MyPageTitle";
```

VTScada will search the application's configuration for a variable named "MyPageTitle" and will use that variable's setting.

```
MyPageTitle = Master Control Page
; entry in Settings.Dynamic
```

	By using DrawLabel, you can modify the title of the page through the configuration variables without needing to access the application's code. This is particularly useful in situations where standard page names must be translated into another language, or otherwise customized for a particular application. Note that many of VTScada's labels are configured in the same way; a variable in the code points to a configuration variable that in turn provides the text for the label. If the DrawLabel variable is absent from the page's source code file, or if DrawLabel is declared in the page's source code file, but the corresponding configuration variable is absent from the application, the value of the Title variable is used as the default. In the event that the Title variable is absent from the page's source code, then the module name is used as the text for the page's title.
Title	The display title for the page. May also be set as an expression within the body of the page's main state.
PageToolTipLabel	Text to display as a tool-tip when the page has been added to the navigation bar.
Bitmap	Name of the image to use for the page background.
NoStretch	The NoStretch variable is associated with the ScaleDisplayContent configuration variable, both of which enable you to control scaling for the pages comprising your application. When set to a non-zero value, ScaleDisplayContent causes

the graphics on all system pages to scale to fit the dimensions of each page. ScaleDisplayContent affects all pages in an application; however, there may be selected pages to which you do not wish the scaling to apply. The NoStretch variable enables you to inhibit scaling for such pages. If NoStretch has a non-zero value, then the page will not be scaled regardless of the setting of the ScaleDisplayContent configuration variable.

Note: Automated display scaling works reasonably well when enlarging the page. It cannot do as good a job when shrinking a display for a smaller screen. In particular, labels embedded within buttons or widgets are more likely to be truncated than scaled down.
Always design for the smallest screen that the application will be displayed upon.

PageWinOpt

The PageWinOpt variable overrides the normal options used for windowed pages (see the Window function for details). The default value for PageWinOpt is 0b1010000100110011.

PageX

The PageX variable enables you to set the X coordinate for the top left corner of a windowed page. If PageX is not set for a windowed page, a default value is used.

PageY

The PageY variable enables you to set the Y coordinate for the top left corner of a windowed page. If PageY is not set for a windowed page, a default value is used.

PageHeight

The PageHeight variable overrides the normal PageHeight calculation for a windowed page (see the Window function for details). If not specified, the height is calculated from the page components.

PageWidth	As above, but for width.
PageMinHeight	Minimum number of pixels to use for the height when the page is displayed in its own window
PageMinWidth	Minimum number of pixels to use for the width when the page is displayed in its own window
PageVWidth	Maximum width, in pixels, of a windowed page
PageVHeigh	Maximum height, in pixels, of a windowed page
PageBMPMarginsWin	A Boolean value (1 or 0) controlling whether a margin should be used when a windowed page displays an image.
PageBMPMarginLeft	Left margin to use when PageBMPMarginsWin is enabled.
PageBMPMarginBottom	Bottom margin to use when PageBMPMarginsWin is enabled.
PageBMPMarginRight	Right margin to use when PageBMPMarginsWin is enabled.
PageBMPMarginTop	Top margin to use when PageBMPMarginsWin is enabled.

Display Manager Bit Flags for Page Display

In addition to the modifiable variables within a page's source file, the Display Manager defines and uses a set of bit flags that determine how a VTScada page is displayed.

```
Constant PSTTB = 0x0001 { Page Style - Show Title Bar };
Constant PSBMP = 0x0002 { Page Style - Show Title Bar Bitmap };
Constant PSLGN = 0x0004 { Page Style - Show Title Bar Logon Button };
Constant PSCFG = 0x0008 { Page Style - Show Title Bar Configure But-
ton };
Constant PSDTE = 0x0010 { Page Style - Show Title Bar Date & Time };
Constant PSIND = 0x0020 { Page Style - Show Title Bar Alarm indic-
ators };
Constant PSTTT = 0x00FF { Page Style - Show all Title Bar decorations
};
Constant PSMBR = 0x0100 { Page Style - Show Task Bar };
Constant PSMNU = 0x0200 { Page Style - Show Menu Button and Menu };
```

```
Constant PSMPB = 0x0400 { Page Style - Show Task bar Page buttons };
Constant PSMFB = 0x0800 { Page Style - Show Task bar "<" and ">" but-
tons};
Constant PSMPM = 0x1000 { Page Style - Show Task bar "+" and "-" but-
tons};
Constant PSMHD = 0x2000 { Page Style - Hold page btn changes target
};
Constant PSMMM = 0xFF00 { Page Style - Show all Menu Bar decorations
};
```

There is a public variable, PageStyle, defined in the Graphics module in each session, which is an OR of the style bits that apply to a page. The initial setting is PSMMM + PSTTT, which is all decorations.

The address of the PageStyle variable is passed as a parameter to the MenuBar and TitleBar plug-ins.

Although PageStyle is public (because the MenuBar and TitleBar plug-ins need to see it) setting its value directly is ineffective because it is reset with each page change.

Whenever a new page is displayed, PageStyle is set as follows (in priority order)…

Normal page:

1. Default value of a PageStyle variable in the page.
2. The value of DefaultPageStyle.

Windowed page:

1. Default value of a PageWStyle variable in the page.
2. The values supplied in parameter 4 of the Display Manager method, ShowStyledPage.
3. The value of DefaultPageStyle.

DefaultPageStyle defaults to the value that shows all decorations. This can be overridden by the configuration settings \DispMgrWPageStyle and \DispMgrPageStyle, or by parameter three of the DisplayManager method ShowStyledPage.

For example, to disable in all pages, the feature whereby a navigation bar button changes if you hold it down for more than 1.5 seconds, you have to set the configuration variable \DispMgrPageStyle to the value 0xFDFF (you cannot use the constant values defined in DisplayManager).

Application properties for pages

- DispMgrPageStyle – For all normal pages
- DispMgrWPageStyle – For windowed pages

(Please refer to Application Properties for the Display Manager " for further information on these and other modifiable variables.)
For Custom pages you can use code similar to the following example to offset the top and bottom of the page in the Display Manager:

```
TopOffset = PickValid(And(Caller(Self())\PageStyle, \Dis-
playManager\PSTTB), 0) ? \DisplayManager\Task_Height : 0;
```

```
BottomOffset = PickValid(And(Caller(Self())\PageStyle, \Dis-
playManager\PSMBR), 0) ? \DisplayManager\Menu_Height : 0;
```

Related Functions:

...Window

...Graphics

...Display Manager Properties – Refer to the VTScada Developer's Guide

---

Create Windows & Use Graphics Functions

You can create user-interface windows using the Window function. An example of a call to the window function can be found in the AppRoot.src file of every new script application.

```
Window(  0,   0           { Upper left corner   },
       800, 600           { View area           },
       800, 600           { Virtual area        },
       Graphics()         { Start user graphics },
        {65432109876543210}
       0b00010000000110011, WinTitle, 0, 1);
```

The Window() function specifies the size, location, and attributes of the window when it is created. The seventh parameter (Graphics() in this example) is a call to start another module. Any graphic statements that appear in the called module or in any of its child modules are drawn on the screen created by the Window function.
If the close icon is enabled on the window (via the Style parameter – 8th parameter), care must be taken to ensure that when the user selects it, the action you desire occurs, as by default, clicking the close button

stops your application; you may prefer to close the window but keep the application running. To help you specify the action you wish to occur when the close button is clicked, the WindowClose() function is provided. For example, you may wish to define a "Cancel" button that will close the window, as well as having the close button enabled:

```
If ZButton(10, 220, 110, 200, "Cancel", 2) || WindowClose(Self())
Done;
[
  ...
  winOpen = 0 { Assuming winOpen was used as the 11th parameter of
Window function };
]
```

When either this cancel button or the close button is selected, the script above will execute, causing the variable that is in the Window function's Enable parameter to become "0", thereby closing the window. Notice that the WindowClose function (as used above), does not actually close the window; rather it checks to see if the user is trying to close the window with the window close button. It returns a true value when this is the case, and the resulting execution of the script closes the window.

Related Information:

...Owned Windows versus Child Windows

...Native Windows Tooltip Support

...Working with Pages

...Focus ID

...Placing Focus on an Object vs. Selecting an Object

...Reference Boxes for Graphics Modules

...Use Scaling to Position Graphic Objects

Related Functions:

... Window

... WindowClose

... WindowOptions

Best Practices for Graphics

When writing a module that will be used to display graphics, there are several things you should be aware of:

The Focus ID

Every function for drawing graphics includes a FocusID parameter. VTScada does not force you to set a unique value for each ID since there are situations where it is desirable to use one value for several objects. (For example, disabling several AddressEntry fields by setting their FocusID values to 0.)

Aside from a few special cases, it is strongly advised that you *do* set a unique FocusID value for each graphic object in a window. Doing so will help ensure that user interface controls function smoothly.

Note: Tab order between user input controls follows their z-order (that is, the order of the statements within the state), rather than their Focus ID value.

The Reference Box

If a particular module is to be used to draw graphics, and this module will be used inside of a transform at any point, it may be helpful to fix the module's reference box size. In doing so, all scaling done by the transform will be predictable. This can be done by using the SetModuleRefBox statement, or (more commonly) by following the module's name in its definition with a group of constants that define the reference box for the module. For example:

```
<
{====================== System\MyModule ========================}
MyModule
(10, 160, 210, 10)
(
  parm1;
  parm2;
)
MainState[
  ...
]
```

```
{ End of System\MyModule }
>
```

The constants define the left, bottom, right, and top coordinates of the module's reference box respectively.
Reference Boxes in Graphic Modules

Switching Pages Within a Window

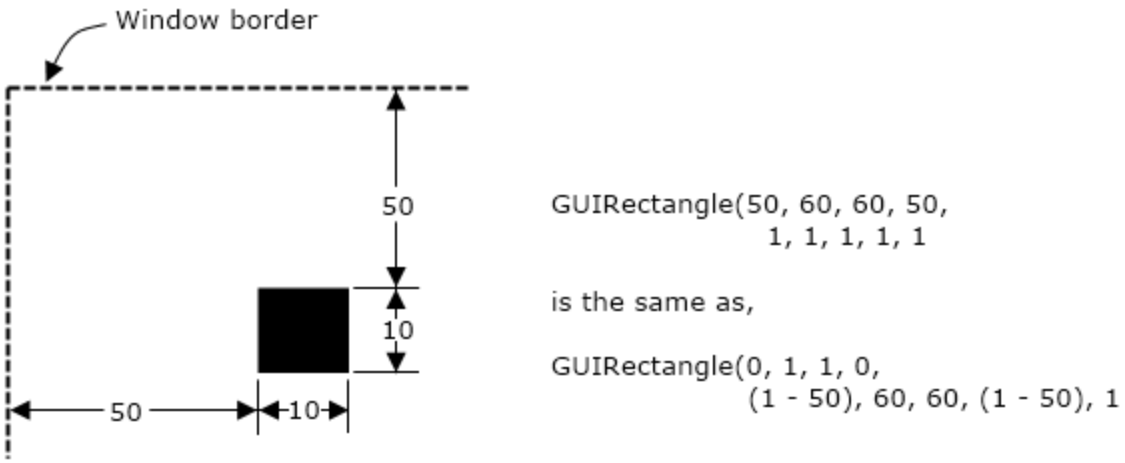
Most applications have multiple graphics. A common example is the multiple tabs of a configuration dialog.
If you want one window to contain several images, the Graphics module should have multiple states, each containing the graphic statements for a given display set. Action-triggers are used to switch from one state to another. This is how tag configuration panels are built.
Another option is to open a separate window for individual pages or dialogs. This can be done by launching a module for each new window using the Launch() function.

Use Scaling to Position GUI Objects

You may want to position an object such as a GUIButton, based on the value of some variable or parameter.
Since the first four parameters of all GUI functions (Left, Bottom, Right, Top) must be constants, the scaling parameters must be used to dynamically change the position and size. This is easier to achieve if the first four parameters define a unit box, and the side-scaling parameters (five through eight) are used to position and size the object being drawn. Since the scaling parameters do not have to be constants, variables may be used to set the object's position.
The first four parameters are always in the order Left, Bottom, Right, Top. The unit bounding box must therefore be defined as 0, 1, 1, 0. Do not change the order.
The scaling parameters are also in the order of Left, Bottom, Right and Top. The scale values to apply will always follow the formula, (1 - Left), Bottom, Right, (1 - Top).

For example, to position a GUIRectangle using the side scaling parameters:

```
left = 10;
bottom = 80;
right = 100;
top = 10;
...
GUIRectangle(0, 1, 1, 0 { Unit bounding box },
             1 - (left) { Left scaling },
             bottom { Bottom },
             right { Right scaling },
             1 - (top) { Top scaling },
             1 { No scaling as a whole },
             0, 0 { No movement },
             1, 0, 0, 0, 0 { Visible, not selectable },
             14, 12 { Yellow interior, red outline });
```



This rectangle will be identical to one drawn using the following constants for the initial bounding box.

```
GUIRectangle(10, 80, 100, 10 { Unit bounding box },
             1, 1, 1, 1, 1 { No scaling },
             0, 0 { No movement },
             1, 0, 0, 0, 0 { Visible, not selectable },
             14, 12 { Yellow interior, red outline });
```

As a final example, suppose that you have created the rectangle depicted in the first case above, and that you now want to draw another rectangle, smaller by 3 pixels in all directions, and perfectly centered within the first rectangle

```
GUIRectangle(0, 1, 1, 0 { Unit bounding box },
             1 - (left + 3) { Left scaling },
             bottom - 3 { Bottom },
             right - 3 { Right scaling },
             1 - (top + 3) { Top scaling },
```

```
            1 { No scaling as a whole },
            0, 0 { No movement },
            1, 0, 0, 0, 0 { Visible, not selectable },
            10, 0 { Green interior, black outline });
```

Again, simply substitute the appropriate scaling coordinates into the formula in the positions held by left, bottom, right and top.

Transparent and Alpha-blended Windows

VTScada supports transparent windows and alpha-blended (also referred to as translucent) windows.

Transparent Windows

A transparent window is one that has a transparent or invisible frame and background. The intended use for this feature is to allow the display of non-rectangular windows by rendering the background as transparent, while allowing mouse messages, such as movement and clicks, to "drill-through" the transparent area to the window below. Objects (such as graphics) placed on the transparent window remain opaque and mouse messages do not drill-through these opaque objects.

The simplest way to create a transparent window is to specify a background color less than zero (e.g. -1). The underlying implementation, however, requires that a specific color be used as the "key" color. All pixels of that color in the otherwise rectangular screen area that the window occupies are rendered as transparent. Using a background color of <0 results in the key color being black (i.e. RGB(0,0,0)), and the window background being set to that color. It may well be that you wish to have black as a color in your window, however. Therefore, setting bit 18 in the Window statement also announces that this window is to be rendered as transparent, with whatever color is specified as the background color being the transparent color.

Alpha-blended or Translucent Windows

An alpha-blended or translucent window is one that has an alpha channel set up in the final renderer, resulting in a translucent effect to the window (i.e. it behaves like a normal window, except that you can see

through the window to some degree). The degree of translucency ranges from 0 (invisible) to 255 (completely opaque, like a normal window). Setting bit 17 on your Window statement invokes "automatic" alpha blending, where the window is set to be 50% translucent when inactive, and opaque when active. This is useful for dialogs that are non-modal and always on-top, so that when another window is active, you can see through the underlying windows. If you need a finer degree of control, do not set bit 17; rather use a new value for the Option parameter of WindowOptions (9) and set the WindowOptions "OptValue" parameter to the degree of alpha-blending that you wish (0 to 255).

Note: Neither the transparent nor the alpha-blended/translucent effects work with child windows. These effects are not designed for animation purposes, and are not sufficiently efficient for this purpose. The amount of processing power required to redraw one of these windows depends on the rating of the graphics card in your machine - the newer and faster the better.

Owned Windows versus Child Windows

Child windows (those with bit 9 set in their Window call) are not recognized as separate entities. Clicking on a child window returns the object value of the root module in its parent window.
A child window is embedded in a parent window and cannot leave that draw area. It is automatically moved when the parent is moved. The child window's X,Y position is relative to the parent's, rather than to the screen. Typically a child window has no frame or title (ie; caption) bar, although you can configure them.
This is not true for owned windows (those with bit 15 set in their Window call), which return the object value of the root module instance in the window.
An owned window is similar to a full, resizable, normal window, however it is owned by another window. Owned windows have no icon on the MS Taskbar. An example of an owned window is the Add Application Wizard.

Owned, caption-less windows are a common way to create pop-up (right-click) menus.
Note that, the mouse-wheel system will treat owned windows the same as child windows only if the former are caption-less.

Native Windows Tooltip Support

The statement, WinTooltipCtrl, supports Windows tooltips.
A Windows tooltip is a pop-up text window that provides an operational hint to a user based on the object on which the mouse pointer is focused. As a VTScada developer, you may choose to present the text box-style tooltips to users, or use the balloon-style tooltips (default). The default behavior is to display balloon-style tooltips for "drawn" objects (for example, an analog input represented as a number on a page), and to display rectangular tooltips for other objects, such as buttons in a tool bar (rectangular tooltips appear for each of the buttons in the tool bar on the Historical Data Viewer page). This behavior can be changed to always display rectangular tooltips by setting the application property "NoBalloonTips" to a non-zero value.
Tooltips in a VTScada application will use a font tag named TipFont *if you have defined one* (you may select any font when defining TipFont). If not defined, the default system font will be used.
The names of the related application properties start with "Tip" (e.g. TipOn & TipFont). The default (Invalid) for all of these settings results in the default operating system settings being used. Please refer to "Application Properties for Tooltips".
The WindowOptions statement allows for setting the text color, background color, and timings associated with all tooltips for a given window. These settings are inheritable, so that Windows that are children of a window that already has these settings modified uses the parent window's settings, unless explicitly overridden by subsequent WindowOptions statements.

Related functions:

... WinTooltipCtrl

... WindowOptions

Working with Pages

Every page in an application will have a matching source file in the Pages sub-folder.

> Note: Developers may choose to remove all source code files before distributing an application to a client.

A page file is a **module**[1]. Using the information in this reference, you can change the appearance and behavior of the pages in your application. There are two levels of access to the characteristics of pages. At the developer's level, page characteristics are set using the Pages library and its associated Add Page and Page Properties dialogs. At the programmer's level, page characteristics can be set within a page's source file.

> Note: Use care if layering custom code on top of existing page objects. (For example, taking a Page Close button and adding your own extra tasks before it executes its own page-closing code.) This may easily result in a race condition. It is better practice to create dedicated code for each task instead.

Page Module / File Characteristics

Page Resources

When a page is created, its resource files are stored within your application directory in a directory named, "Pages". There are three resource files for a page, each with a different extension:

---

[1] A collection of states, scripts, variables, parameters, comments and possibly other modules, all of which make up a VTS program. Modules are separate tasks that run simultaneously in an application. Behind every page is a module and behind every tag on that page is an instance of another module (usually with submodules controlling separate tasks).

- .SRC The source code for the page.
- .RUN The last compiled version of the page.
- .BAK The backup of the source code for the page, after the last editing.

Variables within a Page's Source File

The following variables can be used to modify the attributes and behavior of a page.

- **DrawLabel** When used within a page's source code, the DrawLabel variable identifies the name of a configuration variable whose value provides the text to be used as the page's title. For example, when DrawLabel is declared in the page's source code file as follows:

```
CONSTANT DrawLabel = "PageTitle";
```

  Then VTScada will search the application's configuration for a property named "MyPageTitle" and will use that property's value. This can be useful if the title is to be translated since only a configuration file need be changed rather than the page source code. Many of VTScada's labels are configured in the same way; a variable in the code points to a configuration variable that in turn provides the text for the label.
- **Title** If the DrawLabel variable is absent from the page's source code file, or if DrawLabel is declared in the page's source code file, but the corresponding configuration variable is absent from the application, the value of the Title variable is used as the default. In the event that the Title variable is absent from the page's source code, then the module name is used as the text for the page's title.
- **NoStretch** The NoStretch variable is associated with the ScaleDisplayContent configuration variable, both of which enable you to control scaling for the pages comprising your application. When set to a non-zero value, ScaleDisplayContent causes the graphics on all system pages to scale to fit the dimensions of each page. ScaleDisplayContent affects all pages in an application; however, there may be selected pages to which you do not wish the scaling to apply. The NoStretch variable enables you to inhibit scaling for such pages. If NoStretch has a non-zero value, then the page will not be scaled regardless of the setting of the ScaleDisplayContent configuration variable.

Note: Automated display scaling works reasonably well when enlarging the page. It cannot do as good a job when shrinking a display for a smaller screen. In particular, labels embedded within buttons or widgets are more likely to be truncated than scaled down. Always design for the smallest screen that the application will be displayed upon.

- **PageX** Used to set the X coordinate for the top left corner of a windowed page. If PageX is not set for a windowed page, a default value is used.
- **PageY** Used to set the Y coordinate for the top left corner of a windowed page. If PageY is not set for a windowed page, a default value is used.
- **PageWinOpt** Overrides the normal options used for windowed pages (see the Window() function for details). The default value for PageWinOpt is 0b1010000100110011.
- **PageHeight** Overrides the normal PageHeight calculation for a windowed page (see the Window() function for details). If not specified, the height is calculated from the page components.

Display Manager Bit Flags for Page Display

In addition to the modifiable variables within a page's source file, the Display Manager defines and uses a set of bit flags that determine how a VTScada page is displayed.

- Constant PSTTB = 0x0001 { Page Style - Show Title Bar };
- Constant PSBMP = 0x0002 { Page Style - Show Title Bar Bitmap };
- Constant PSLGN = 0x0004 { Page Style - Show Title Bar Logon Button };
- Constant PSCFG = 0x0008 { Page Style - Show Title Bar Configure Button };
- Constant PSDTE = 0x0010 { Page Style - Show Title Bar Date & Time };
- Constant PSIND = 0x0020 { Page Style - Show Title Bar Alarm indicators };
- Constant PSTTT = 0x00FF { Page Style - Show all Title Bar decorations };
- Constant PSMBR = 0x0100 { Page Style - Show Task Bar };
- Constant PSMNU = 0x0200 { Page Style - Show Menu Button and Menu };
- Constant PSMPB = 0x0400 { Page Style - Show Task bar Page buttons };
- Constant PSMFB = 0x0800 { Page Style - Show Task bar "<" and ">" buttons};
- Constant PSMPM = 0x1000 { Page Style - Show Task bar "+" and "-" buttons};

- Constant PSMHD = 0x2000 { Page Style - Hold page btn changes target };
- Constant PSMMM = 0xFF00 { Page Style - Show all Menu Bar decorations };

There is a public variable, PageStyle, defined in the Graphics module in each session, which is an OR of the style bits that apply to a page. The initial setting is PSMMM + PSTTT, which is all decorations.

The address of the PageStyle variable is passed as a parameter to the MenuBar and TitleBar plug-ins.

Although PageStyle is public (because the MenuBar and TitleBar plug-ins need to see it) setting its value directly is ineffective because it is reset with each page change.

Whenever a new page is displayed, PageStyle is set as follows (in priority order)…

Normal page:

1. Default value of a PageStyle variable in the page.
2. The value of DefaultPageStyle.

Windowed page:

1. Default value of a PageWStyle variable in the page.
2. The values supplied in parameter 4 of the Display Manager method, ShowStyledPage.
3. The value of DefaultPageStyle.

DefaultPageStyle defaults to the value that shows all decorations. This can be overridden by the configuration settings \DispMgrWPageStyle and \DispMgrPageStyle, or by parameter three of the DisplayManager method ShowStyledPage.

For example, to disable in all pages, the feature whereby a taskbar button changes if you hold it down for more than 1.5 seconds you have to set the configuration variable \DispMgrPageStyle to the value 0xFDFF (you cannot use the constant values defined in DisplayManager).

Note: You can incorporate the setting of these flags into the DisplaySession AppSessionVars plug-in module. Using this technique enables the presentation to vary under different conditions. For example, you may wish to present a welcome page which features no taskbar and no alarm indicator in the title bar to users who are not

logged in. Once the users log in, you can then present the page with menu/page navigation and alarm indicators enabled.

Application properties for pages

- DispMgrPageStyle - For all normal pages
- DispMgrWPageStyle - For windowed pages

(Please refer to Application Properties for the Display Manager " for further information on these and other modifiable variables.)

For Custom pages you can use code similar to the following example to offset the top and bottom of the page in the Display Manager:

```
TopOffset = PickValid(And(Caller(Self())\PageStyle, \Dis-
playManager\PSTTB), 0) ? \DisplayManager\Task_Height : 0;
BottomOffset = PickValid(And(Caller(Self())\PageStyle, \Dis-
playManager\PSMBR), 0) ? \DisplayManager\Menu_Height : 0;
```

Focus ID

Every function for drawing graphics includes a FocusID parameter. VTScada does not force you to set a unique value for each ID since there are situations where it is desirable to use one value for several objects. (For example, you may want to disable several AddressEntry fields by setting their FocusID values to 0.)

Aside from a few special cases, it is strongly advised that you *do* set a unique FocusID value for each graphic object in a window. Doing so will help ensure that user interface controls function smoothly.

Note: Tab order between user input controls follows their z-order (that is, the order of the statements within the state), rather than their Focus ID value.

Related functions:

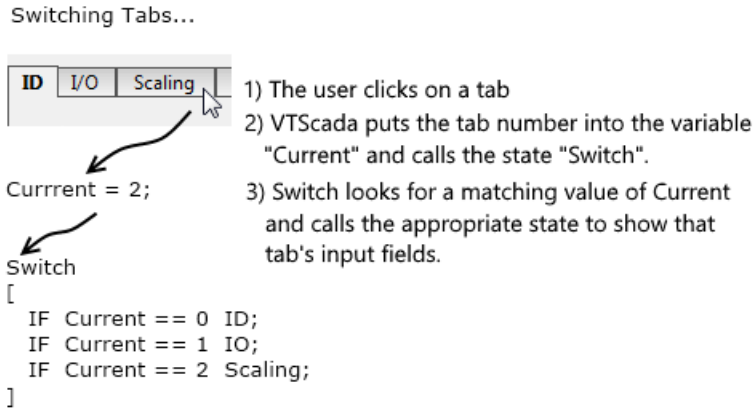
... FocusID

... NextFocusID

...Graphics

Switching Graphics Pages

Most applications have multiple graphics. If you want one window to contain several images, the Graphics module should have multiple states, each containing the graphic statements for a given display set. Action-triggers are used to switch from one state to another. This technique is used by tag configuration panels.



The second alternative is to open a separate window for individual pages or dialogs. This can be done by launching a module for each new window using the Launch function.

Related functions:

... Launch

Placing Focus on an Object vs. Selecting an Object

There is a difference between an object being selected, and an object that has the focus on a system page. It is very important to note that an object that has the focus is not necessarily selected.

An object that has the focus is ready for input from the user. If the object is an edit field for example, the cursor will blink within the field, indicating that it is ready for input. If the object with the focus is a button, it is highlighted when selected. In order for an object that has the focus to be selected, it must be clicked by the mouse, or the <TAB> key, or the <RETURN> or <ENTER> key must be pressed.

It is possible to force the input focus to a certain graphic object by means of a NextFocusID statement. When dealing with statements that

combine an If function and a GUIButton function, it is important to keep in mind that you may focus the button, but the script paired with the If function will not be executed until the button is actually selected (either by the keyboard or the mouse).
Focus movement (on a tab key) or reverse tab (i.e. Shift + Tab keys)) is based on the order that statements appear in the source code. This applies recursively to calling sequences in steady state. For example:

```
ZButton(...1...);
ChildMod();
ZButton(...2...);
```

In this example, ZButton(...1...) would be first in the focus order, followed by any focusable statements in ChildMod, followed by ZButton (...2...).
Launched modules will appear in the focus order after steady state focusable statements and steady-state calling sequences.
For example:

```
If Watch(1);
[
  LaunchedMod();
]
ZButton(...1...);
ChildMod();
ZButton(...2...);
```

In the above sequence, the focus order will be the same as that of the previous example above, with the addition of any focusable statements in LaunchedMod, after ZButton(...2...).

Reference Boxes for Graphics Modules

If a particular module is to be used to draw graphics, and this module will be used inside of a transform at any point, it may be helpful to fix the module's reference box size. In doing so, all scaling done by the transform will be predictable. This can be done by using the SetModuleRefBox statement, or (more commonly) by following the module's name with a group of constants that define the reference box for the module.
Example:

```
<
{==================== System\MyModule ======================}
MyModule
(0, 1, 1, 0)
(
   parm1;
   parm2;
)
MainState[
...
]
{ End of System\MyModule }
>
```

The constants define the left, bottom, right, and top coordinates of the module's reference box respectively. The one-unit values shown in the example are commonly used when the intention is to allow the transform code to control the final size. By using unit values, you greatly simplify calculations of scale.
The same technique is also used by all of the GUI... functions.
Note that these values must be constants; the use of variables for the four values is not allowed by the compiler.

Related information that you may need:

...Use Scaling to Position Graphic Objects

...Reference Boxes in Graphic Modules

Related functions:

... GUIArc

... GUIBitmap

... GUIButton

... GUIChord

... GUIEllipse

... GUIPie

... GUIPipe

... GUIPolygon

... GUIRectangle

... GUIText

... GUITransform

Use Scaling to Position Graphic Objects

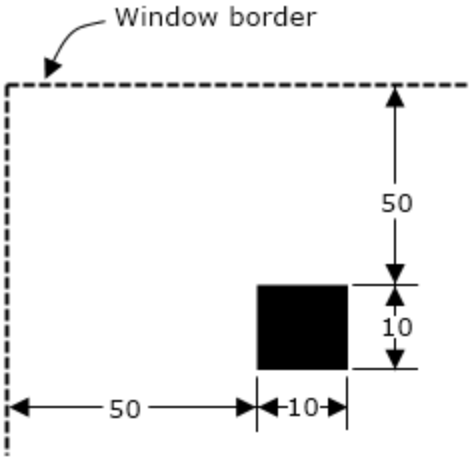
It is sometimes desirable to position an object, such as a GUIButton, based on the value of some variable or parameter.
Since the first four parameters of all GUI functions must be constants, the scaling parameters must be used to dynamically change the position and size. This is easier to achieve if the first four parameters are used to define a unit box, and the side-scaling parameters (five through eight) are used to position and size the object being drawn. Since the scaling parameters do not have to be constants, variables may be used to set the object's position.
The first four parameters are always in the order Left, Bottom, Right, Top. The unit bounding box must be defined as 0, 1, 1, 0. Do not change the order.
The scaling parameters are also in the order of Left, Bottom, Right and Top. The scale values to apply will always follow the formula, (1 - Left), Bottom, Right, (1 - Top).
For example, to position a GUIRectangle using the side scaling parameters:

```
left = 10;
bottom = 80;
right = 100;
top = 10;
...
GUIRectangle(0, 1, 1, 0 { Unit bounding box },
             1 - (left) { Left scaling },
             bottom { Bottom },
             right { Right scaling },
             1 - (top) { Top scaling },
             1 { No scaling as a whole },
             0, 0 { No movement },
             1, 0, 0, 0, 0 { Visible, not selectable },
             14, 12 { Yellow interior, red outline });
```

This rectangle will be identical to one drawn using the following constants for the initial bounding box.

```
GUIRectangle(10, 80, 100, 10 { Unit bounding box },
             1, 1, 1, 1, 1 { No scaling },
             0, 0 { No movement },
```

```
            1, 0, 0, 0, 0 { Visible, not selectable },
            14, 12 { Yellow interior, red outline });
```




```
GUIRectangle(50, 60, 60, 50,
                1, 1, 1, 1, 1

is the same as,

GUIRectangle(0, 1, 1, 0,
                (1 - 50), 60, 60, (1 - 50), 1
```

As a second example, suppose that you have created the rectangle depicted in the first case above, and that you now want to draw another rectangle, smaller by 3 pixels in all directions, and perfectly centered within the first rectangle

```
GUIRectangle(0, 1, 1, 0 { Unit bounding box },
             1 - (left + 3) { Left scaling },
             bottom - 3 { Bottom },
             right - 3 { Right scaling },
             1 - (top + 3) { Top scaling },
             1 { No scaling as a whole },
             0, 0 { No movement },
             1, 0, 0, 0, 0 { Visible, not selectable },
             10, 0 { Green interior, black outline });
```

Again, simply substitute the appropriate scaling coordinates into the formula in the positions held by left, bottom, right and top.

Related Information:

...Reference Boxes for Graphics Modules – General overview

...Reference Boxes in Graphic Modules – Specific details

Related functions:

... GUIArc

... GUIBitmap

... GUIButton

... GUIChord

... GUIEllipse

... GUIPie

... GUIPipe

... GUIPolygon

... GUIRectangle

... GUIText

... GUITransform

Drag & Drop to a Window

Any window can be used as a drag and drop target. An example can be seen in the Idea Studio, where you can drag an image from any Windows folder directly to your editing canvas to both import the image and draw it on the canvas.
Not all object types can be imported.
To add this functionality, you must include two call-back modules in the module that controls the window. These are OLEDrag and OLEDrop. OLEDrag is not strictly necessary, but without it users will have no visual reference that a drag and drop operation is under way.
In the following example, the parameters shown are required to make the call-backs work. The content of the modules is entirely up to you. It is a requirement that these modules operate as subroutines.

Example:

```
{========================== System
================================}
{=================================================================-
===}
( System               { Provides access to system library func-
tions };
  Layer                { Provides access to the application layer
 };
)
[
  Graphics      Module { Contains user graphics
 };
  WinTitle = "IDropTarget Test" { Window title
 };
]
```

```
Main [
  Window(  0,   0           { Upper left corner   },
         800, 600           { View area           },
         1600, 1200         { Virtual area        },
          Graphics()         { Start user graphics },
         {65432109876543210}
          0b00010000000110011, WinTitle, 0, 1);
]

<
{======================== System\Graphics
============================}
{ This module handles all of the graphics for the application
   }
{=================================================================-
=====}
Graphics
[
  OLEDrag        MODULE { Called when a droppable object passes
over    };
  OLEDrop        MODULE { Called when a droppable object is dropped
here};
  PlaceImage     MODULE { Draws an image in the position it was
dropped };
  PROTECTED PlacedImages { Dictionary of dropped image objects
    };
  PROTECTED CurrentDragItem  { Storage for current droppable obj if
any };
  PROTECTED CurrentDragImage { Image being dragged if applicable
    };
  PROTECTED DragX             { Current drag X position
    };
  PROTECTED DragY             { Current drag Y position
    };
  PROTECTED ImgSzX            { Current image width
    };
  PROTECTED ImgSzY            { Current image height
    };
  PROTECTED Base              { This object
    };
  CONSTANT  #CF_TEXT     = 1  { Text clipboard format
    };
  CONSTANT  #CF_HDROP    = 15 { File drop clipboard format
    };
]

Init [
  If 1 Main;
  [
    PlacedImages = Dictionary();
    Base         = Self();
  ]
]
```

```
Main [
  ZText(10, 30,
        "Click and drag MSDN_Butterfly.jpg from the app folder", 0,
0);
  ZText(10, 45,
        "onto this window.", 0, 0);
  ZText(10, 60,
        "Does the image appear and track to the cursor (roughly)",
0, 0);
  ZText(10, 75,
        "when the mouse is over this window?", 0, 0);
  ZText(10, 90,
        "Release the mouse button, dropping the image.", 0, 0);
  ZText(10, 105,
        "Does a copy of the image appear in the dropped position?",
0, 0);
  ZText(10, 120,
        "If the answer to both questions above is yes then this",
0, 0);
  ZText(10, 135,
        "test is a success, otherwise it fails. Feel free to try",
0, 0);
  ZText(10, 150,
        "other files. Only images should be processed by this
test.", 0, 0);
  { The following code draws a copy of the image on the page posi-
tioned
    to match the last reported drag position. The image is drawn
extending
    up and left from the position.
       }
  ImgSzX = BitmapInfo(CurrentDragImage, 0);
  ImgSzY = BitmapInfo(CurrentDragImage, 1);
  GUIBitmap(0, 1, 1, 0,
            1 - (DragX - ImgSzX),
            DragY,
            DragX,
            1 - (DragY - ImgSzY),
            1, 0, 0 { No overall scaling, trajectory, or rotation
},
            1, 0    { visibility, reserved },
            0, 0, 0 { No activation or focus },
            CurrentDragImage);
]

<
{============================== OLEDrag
==============================}
{ Called in response to an IDropTarget drag notification.
  }
{=================================================================-
====}
OLEDrag
(
  Type                { The clipboard format of the data partameter
```

```
  };
  Data                { The data passed, data type varies (see
above) };
  KeyState            { Keyboard key press enumeration
  };
  X                   { X-coordinate of the cursor
  };
  Y                   { Y-coordinate of the cursor
  };
  Mode                { Op code: 0=drop, 1=enter, 2=over, 3=exit
  };
)

Main [
  If 1;
  [
  { When a drag enters a window it reports the data, but does not
during drag over or exit operations. Grab a copy of the image dur-
ing the enter and
    remove it upon exit.
         }
  IfElse(Mode == 1 && Type == #CF_HDROP, Execute( { Only try to
make a bitmap if given a file name }
    CurrentDragItem = Data;
    CurrentDragImage = MakeBitmap(CurrentDragItem);
  );
  { Else } IfThen(Mode == 3,
    CurrentDragItem = CurrentDragImage = Invalid;
  ));
{ Update the drag position, technically this is only necessary
because
  XLoc etc. don't report mouse position during a drag.
   }
  DragX = X;
  DragY = Y;
  Return(Invalid);
]
]
{ End of System\Graphics\OLEDrag }
>

<
{============================== OLEDrop
==============================}
{ Called in response to an IDropTarget drop notification.
   }
{=================================================================-
=====}
OLEDrop
(
  Type                 { The clipboard format of the data par-
tameter   };
  Data                 { The data passed, data type varies (see
above) };
  KeyState             { Keyboard key press enumeration
```

```
   };
  X                    { X-coordinate of the cursor
   };
  Y                    { Y-coordinate of the cursor
   };
  Mode                 { Op code: 0=drop, 1=enter, 2=over, 3=exit
   };

)
[
  PROTECTED Image      { Image loaded from the file
   };
]

Main [
  If 1;
  [  { The imae is being dropped, cease to draw the drag-tracking
image }
    CurrentDragItem = CurrentDragImage = Invalid;
    { Add a copy of the image to the window at the current coordin-
ates. }
    IfThen(Type == #CF_HDROP { Only do this if we were passed a
file name },
      Image = MakeBitmap(Data);
    IfThen(Valid(Image),
      PlacedImages[GetGUID(1)] = Launch(\PlaceImage, Base, Base,
Image, X, Y);
    );
    );
    Return(Invalid;);
  ]
]
{ End of System\Graphics\OLEDrop }
>

<
{========================== PlaceImage
==============================}
{ Draws an image on the window at with a lower right corner at the
given pos. }
{=================================================================-
=====}
PlaceImage
(
  Image                { Image to be placed
   };
  X                    { Right side of the placement
   };
  Y                    { Left side of the placement
   };
)
[
  PROTECTED ImgSzX     { Current image width
   };
  PROTECTED ImgSzY     { Current image height
   };
]
```

```
Main [
{ Draw an image extending up and left from the given position. This
just
  makes things easy given the way that the image is being traced
during the
  drag.
       }
  ImgSzX = BitmapInfo(Image, 0);
  ImgSzY = BitmapInfo(Image, 1);
  GUIBitmap(0, 1, 1, 0,
            1 - (X - ImgSzX),
            Y,
            X,
            1 - (Y - ImgSzY),
            1, 0, 0 { No overall scaling, trajectory, or rotation
},
            1, 0    { visibility, reserved },
            0, 0, 0 { No activation or focus },
            Image);
]
{ End of System\Graphics\PlaceImage }
>
{ End of System\Graphics }
>
```

TreeControl Module

The tree control is a system-level tool in VTScada. The TreeControl module implements a tree control similar to that used by the Microsoft Windows Explorer folder panel.

The format for the TreeControl module is:

```
TreeControl(&Tree)
```

where &Tree is a reference to the tree to be displayed.

TreeControl makes the following constants available to any module that calls the ImportAPI function:

```
{***** Indices into the Tree array nodes *****}
[ (API)
  Constant #TI_KEY         = 0 { "Key" value...see heading comment };
  Constant #TI_TEXT        = 1 { Text value to be displayed        };
  Constant #TI_SUBTREE     = 2 { Subordinate tree below this node  };
  Constant #TI_MAPARRAYIDX = 3 { MapArray entry index              };
  Constant #TI_FLAGS       = 4 { Various flags...see definitions   };
  Constant #TI_ICON        = 5 { ICON graphic for node..folder is
                                 default };
  Constant #TI_TOOLTIP     = 6 { In-place tooltip text - #TI_TEXT
                                 default };
  Constant #TREE_MINNODESIZE = 7 { Minimum compulsory node size    };
  Constant #TIF_EXPANDED   = 1 { Flag - true if expanded           };
  Constant #TIF_CANEXPAND  = 2 { Flag - true if able to expand     };
```

```
  Constant #TIF_NOFOLDER   = 4 { Flag - true if not display folder
                                 bmp  };
  Constant #TIF_GREYTEXT   = 8  { Flag - true if grey this option  };
  Constant #TIF_HIDDENROOT = 16 { Flag - true if root is a hidden
                                 root   };
  Constant #TIF_TITLEDTIP  = 32 { Flag true to use titled tooltips };
  Constant #TIF_POPUPTIP   = 64 { Flag - true to NOT use in-place
                                 tooltips };
]
```

The caller (not the parent) can provide the following subroutine modules that will be called by TreeControl in response to specific events:

```
OnLeftClick(Node, X, Y)
```

Called when the left mouse button is released over a tree node. Node is the tree node, while X and Y are the coordinates of the mouse.

```
OnRightClick(Node, X, Y)
```

Called when the right mouse button is released over a tree node. Node is the tree node, while X and Y are the coordinates of the mouse.

```
OnDoubleClick(Node, X, Y)
```

Called when the left mouse button is double-clicked over a tree node. Node is the tree node, while X and Y are the coordinates of the mouse. This callback is always proceeded by OnLeftClick() and any node expansion is done prior to calling OnDoubleClick(), but after OnLeftClick().

```
CreateSubtree(Node)
```

Called when a tree node has its #TIF_CANEXPAND flag set, but the #TI_SUBTREE member of the node has not yet been constructed. The callee is expected to construct an array of nodes and store them in the node supplied to CreateSubtree.

```
ExpandTreeToNode(Key)
```

A sort of superset of CreateSubtree. It is called in response to a call to SetSelected() to command the caller of TreeControl to make all the tree nodes necessary to allow the node containing the Key to be expanded. The caller of TreeControl should call ExpandNode as needed for each node. When this callback returns, the tree will be positioned at the node that contains Key.

The (rough) logic of ExpandTreeToNode is:

Recursively walk up the tree by recursing this subroutine until you get to the tree root. The reverse recursion path is the shortest route from the root back to the node. Unwind the recursion, creating sub-trees as necessary and calling ExpandNode() for any that are not expanded.
Clicking on the junction is handled internally, and no callback is made.
The array that is passed in describes the tree structure. The array must be a 2-dimensional array, with each row (first subscript) describing a node at the same level in the tree. Each field in the row describes the node further:

```
[n][#TI_KEY]
```

The Key value is user-defined, and must be a value for which the == operator is meaningful. The Key value is used to identify which node the caller is talking about when calling helper subroutines.

```
[n][#TI_TEXT]
```

This is the text value displayed. It can be any VTScada value that has a valid textual representation.

```
[n][#TI_SUBTREE]
```

This is a reference to an array of subordinate nodes that are of the same format as this node.

```
[n][#TI_FLAGS]
```

Flag values are used internally.

```
[n][#TI_ICON]
```

ICON graphic for the node (a folder graphic by default).

```
[n][#TI_TOOLTIP]
```

Additional tooltip for the node (none by default).
You can have each row with as many elements as you wish, but the above indices are reserved and must be present in all nodes. The "structured" Tree array provided can be modified at any time, and the TreeControl will faithfully follow it. You can call Refresh at any time to invoke a full rebuild of the tree.

Related Functions:

... GridList

... ImportAPI

Time and Date

Within VTScada, the time and date are kept as two separate numbers. The time of day is represented as the number of seconds since midnight, while the date is represented as the number of days since January 1, 1970.

Units for Time:

The Seconds function returns the time, the Today function returns the date, and the CurrentTime function returns a combination of the two in the form of the number of seconds since January 1, 1970.

Seconds returns a double value slightly more accurate than 1 micro-second; however, when that value is assigned to a float type variable, it is rounded to about 7 significant digits. This can reduce the accuracy, especially later in the day when the seconds count becomes large. To obtain a reasonably accurate time stamp, the expression Seconds % 1 can be used to return fractions of a second since the last second mark.

These date and time numbers may be manipulated to determine elapsed time between two events simply by taking the difference between the time and date of the two events. One of the more effective ways to accomplish this is by using the CurrentTime function. You can use the single date and time number that is returned by this function to easily calculate the elapsed time between events or to calculate the date and time at a fixed offset.

The date and time numbers can be converted into text values for display purposes using the Date and Time functions. The values returned from these functions may have a variety of formats and may be displayed on the screen using a GUIText statement. The Now function can be used in a

statement to give the time suitable for a Time function, or to display a simple clock.
Numeric values of the day, month, and year may be extracted from a VTScada date value (number of days since January 1, 1970) using the Day, Month, and Year functions. The reverse process of combining the day, month, and year may be done using DateNum.

Related information you may need:

...VTScada Time Zones

...Timers and Timing

Related Functions:

...Time And Date

VTScada Time Zones

VTScada provides three time zone functions:

- TimeZoneList(): The TimeZoneList() function provides a list of time zones.
- ConvertTimestamp(): The ConvertTimestamp() function converts a timestamp between different time zones.
- TimeZone(): The TimeZone() function returns information on the current time zone, such as the time zone name.

Unfortunately, TimeZoneList() and ConvertTimestamp() use information in the registry that is not localized on non-English versions of Windows operating systems; however, TimeZone() does. This causes problems when you need to convert a timestamp to or from the local time zone, as the result of TimeZone(2) is not suitable as a parameter to ConvertTimestamp() on non-English systems.
To overcome this issue, a time zone of "0" (local time zone) may be specified to ConvertTimestamp() as either the source or destination time zone.

Timers and Timing

There are three timer functions that can be used to indicate when a specified time period has elapsed.

- AbsTime is used to check when a certain time occurs (relative to the real time clock). For example, AbsTime can test when the next time a shift change will occur. Since this function is tied into the real time clock, it is not subject to drift as is the TimeOut function. It is useful when you want to tie an event to a particular time of day rather than to a fixed time delay. If the absolute time of the event is important, AbsTime should be used; however, if the time of the event is meant to be relative to a randomly occurring event, TimeOut would be the most appropriate function to use.
- RTimeOut is similar to TimeOut, except that it remembers the elapsed time accumulated so far, even when the enable parameter is false. For example, it can be used to test when a certain piece of equipment reaches a specific total cumulative running time.
- TimeArrived indicates whether a given time, provided as a timestamp, has occurred.
- TimeOut returns a value of "true" only after a fixed period of time has elapsed. This function is often used in action triggers to cause an event to occur after a time delay.

Related Functions:

...AbsTime

... RTimeOut

... TimeArrived

... TimeOut

Build Custom Reports

You can build your own custom report-types that integrate into the VTScada report page or report tag, and therefore make use of the VTScada Report Page's features (selection of tags, start and end dates, output formats). Your module can then control the data retrieval process, defining what is retrieved and what further calculations are done on the returned values.

Related information you may need:

How Reports Collect Data

All reports make use of the GetTagHistory(1) function to query logged data. This function returns an array of values from each tag, according to the query parameters. These values are then formatted for display in the report.

The features that make one report differ from another include:

- The title.
- The selection of tags.

Each report provides its own filter to the report page to limit the type of tags available for selection.

- The data to return from a tag.

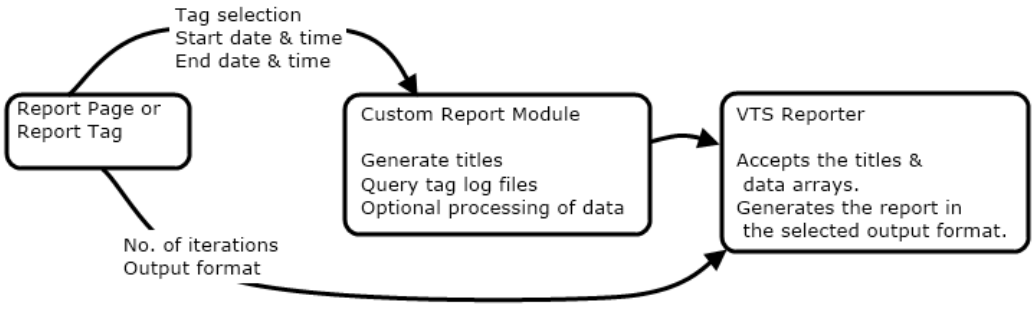
In almost all cases, this is "Value" - the logged values from a tag. But, custom tags can contain fields other than Value that can be logged and therefore reported on.

- The start and end dates.
- The time range per data point retrieved.

For example, you might query the set of maximum daily values during a month. The start and end dates mark the beginning and end of the month, while the time range per retrieved data point will be one day.

- A function to be applied to the data in each time range.

Eleven functions are available including time-weighted average, minimum in range, maximum in range, sum of zero to non-zero transitions during the range, etc.



[1]GetTagHistory replaces the older function GetLog. GetLog is still supported. Legacy applications that made use of GetLog can continue to do so.

Report Formatting

VTScada reports are designed so that they can be sent to a range of output formats including a text document, the system printer, an email or an Excel spreadsheet. For this reason, they are designed with a minimum of complicated formatting. Each report will contain a title (usually including the date range), rows and columns of data, and very little else.
Writing VTScada code for more complex formatting is not recommended. If a fancier report appearance is required, you can copy the default output into a document or spreadsheet and apply fonts, colors, cell borders, sub-total breaks, and other features there. Macros in programs such as Excel can help automate this process if it needs to be done on a repeating basis.
By following this approach, you ensure that your reports remain flexible for use with a variety of output formats.

Common Features of a Report Module

All report modules must contain the following features:

- Module structure follows VTScada coding rules.
- Parameter list enables the Report Page to pass user selections in.

Shared reference to Report so that the module becomes a plug-in of the Report Page.

- Variables include:
- TypeFilter – limits tag selection in the Report Page
- ReportName – displayed Report Type in the Report Page
- The initialization state sets the query parameters and calls GetLog for each user-selected tag.
- A second state processes the data, sending the rows to the VTScada report generator, then slays the module when finished.

These features can be seen in the following example. This module creates a snap-shot type report. Instead of collecting an hourly or daily snapshot, it retrieves the value at the start of each 15-minute data range.

You can add this report to any VTScada application by copying the code into a text file and declaring it in the AppRoot.SRC of the application. For example, if you named the file containing this module, "QuarterHour.src" declare it in the (PLUGINS) section of the AppRoot.SRC file as follows:

```
[(PLUGINS)
    QuarterHour   Module   "QuarterHour.src";
]
```

Remember that file changes must always be imported before they come into effect.

The code for the example follows:

```
{============================ QuarterHour
================================}
{ This generates a snapshort report given the time interval
}
{=================================================================-
=====}
(
  Reporter                { Object value for call-backs
};
  Start                   { Starting time (local, not UTC!)
};
  End                     { Ending time
};
  Tags                    { List of tag names to report on
};
)

[
  Titles                  { Array of ODBC database field names
};
  Types                   { Array of ODBC column type names
```

```
};
  Data                    { Actual data for the body of the report
};
  Format                  { Array of format strings which will be
used
                            in successive SWrites to write Data
};
  TitleStrm               { Stream to build title line in
};
  i                       { Loop counter
};
  j                       { Loop counter
};
  NTags                   { Number of tags
};
  NRows                   { Number of periods (rows) in the report
};
  TagData                 { Array of tag data
};
  TZBias                  { Only Valid if Called from Report Page
};

  Constant NUMFMT = "%13.2f";
  Constant COLFMT = "%13s";

    { Set up this module to become a plug-in for the reports }
  [(POINTS)
    Shared Report;
  ]

  Constant TypeFilter = "Loggers"     { type of tags to use in the
report };
  Constant ReportName = "Quarter Hour Snapshot" { title for the
report            };
  Constant TPP        = 900             { 900 sec = 15 minutes
};
  Constant Mode       = 4               { Mode 4 == Value at start of
TPP };
                          { The mode value of 4 is what makes this
a
                             Snapshot-type report.               }
]

Init [

  If 1 Loop;
  [
      { Initialize the arrays for the tags }
    TZBias  = \IsTimeZoneAware ? TimeZone(0) : Invalid;
    NTags   = ArraySize(Tags, 0);
    TagData = New(NTags);

      { Remaining arrays include Date and Time as well as NTags }
    Titles  = New(NTags + 2);
    Types   = New(NTags + 2);
    Data    = New(NTags + 2);
    Format  = New(NTags + 2);
```

```
    { Build the title lines }
   TitleStrm = BuffStream(0);
   SWrite(TitleStrm, "%s from %s %s to %s %s\r\n\r\n",
          ReportName { Title for the report },
          Date(Start / 86400, 4), Time(Start % 86400, 2),
          Date((End - TPP) / 86400, 4),
          Time((End - TPP) % 86400, 2));

    { Set the value, type and format of the first two title columns
}
   Titles[0] = "Date";
   Titles[1] = "Time";
   Types [0] = "TEXT";
   Types [1] = "TEXT";
   Format[0] = "%-13s";
   Format[1] = "%-9s";

    { write the first 2 columns of the title line }
   SWrite(TitleStrm, Format[0], Titles[0]);
   SWrite(TitleStrm, Format[1], Titles[1]);

    { Reset the loop counter }
   i = 0;
    { For each selected tag... }
   WhileLoop(i < NTags;
      { Add the tag name to the title line }
     Titles[i + 2] = PickValid(Scope(\Code, Tags[i])\Name,
"Unknown");
     Types [i + 2] = "TEXT";
     Format[i + 2] = COLFMT;
     SWrite(TitleStrm, COLFMT, Titles[i + 2]);

      { and, query data for the tag }
     \GetLog(&TagData[i],
             Scope(\Code, Tags[i]) { Point object value      },
             "Value"               { Read data               },
             Start                 { Start time              },
             End                   { End time                },
             TPP                   { Time per point          },
             Invalid               { No max number of points },
             Mode                  { Calculation mode        },
             Invalid               { N/A                     },
             Invalid               { Stale time              },
             TZBias                { Time Zone Bias          });
     i++;
   );

     { Add CR, LF to title line and reset the stream }
   SWrite(TitleStrm, "\r\n");
   Seek(TitleStrm, 0, 0);

     { Let the VTScada Reporter module set up the ODBC columns }
   Reporter\ODBCColumns(Titles, Types);
     { And, the title line }
   Reporter\TitleLine(TitleStrm);
 ]
```

```
]

Loop [
   { ensure that GetLog has finished }
  If AValid(TagData[0], NTags) == NTags;
  [
     { Calculate the correct number of rows to avoid off-by-one
errors }
    NRows = Ceil((End - Start) / TPP);

     { Reset the row counter }
    j = 0;
     { Loop through retrieved data, creating each report row }
    WhileLoop(j < NRows;
       { First two columns of the row will be date and time }
      Data[0] = Date((Start + j * TPP) / 86400, 7); { Date at start
of TPP }
      Data[1] = Time((Start + j * TPP) % 86400, 2); { Time at start
of TPP }
       { Loop through TagData array to get the data }
      i = 0;
      WhileLoop(i < NTags;
         { Fill in data array     }
        Data[i + 2] = Valid(Cast(TagData[i][j], 3)) ?
                      TagData[i][j] : Invalid;
         { Fill in format array   }
        Format[i + 2] = NUMFMT;
         { Increment loop counter }
        i++;
      ); { End WhileLoop          }

       { Pass the report line to the VTScada Reporter module }
      Reporter\DataLine(Format, Data);
       { Increment the data time index        }
      j++;
    ); { End WhileLoop }
    Slay(Self, 0);
  ]
]
```

In this example, each call to GetLog returned a 1-dimensional array. If you had passed it an array of modes, or an array of fields to return (perhaps TimeStamp and Value), then the result from each call would be an array of corresponding dimensions.

Type Filters - Limiting the List of Available Tags

By declaring a constant named "TypeFilter" you can limit the range of tags available to the user for selection in the VTScada Reports Page. Possible values include any tag name or group. Since only logged data is available to be reported on, it is common to set this to "Loggers" but for your

report it may be appropriate to limit the selection to "Pump Status" or "Digitals".

Example:

```
Constant TypeFilter = "Loggers";
```

Parameters in a Custom Report

Five parameters should be declared in your module. In addition to the VTScada Reporter object, these allow user-selected parameters from the VTScada Report Page to be passed to your custom report. The standard parameters are as follows:

- Reporter
  Object value for call-backs.
- Start
  User-selected starting time.
- End
  User-selected ending time.
- Tags
  User-selected list of tag names to report on

In addition to these, you may decide to create a generic report that can be used for a number of variations. The VTScada Snapshot report is an example: That report module takes several extra parameters, which may be overridden by a calling module, thereby creating different types of Snapshot report. As a suggestion, extra parameters may include:

- Vars
  List of logged variables within tags. This will always be "Value" when reporting on VTScada tags, but if your application contains custom tags with other logged variables, you may override this parameter to provide the variable name, or an array of logged variable names.
- TimeRange
  Time range in seconds for each data point retrieved by the query. Passed to the GetLog function as the TPP parameter. Appropriate choices for this value will depend on the selected mode.
- ReportName

For customized report naming.

- Mode
  The Mode to run GetLog in, thereby changing how data is queried. Please see the following topic, Query Modes and Time Ranges.
- StaleTime
  Optional: used in Mode 11 (Rollover Totals). See GetLog in the function reference for more details.

Query Modes and Time Ranges

When combined, these two parameters determine what report will be generated from a given set of tags. The choice of Mode determines how the raw data will be retrieved. The choice of time range selects the amount of data included in each of Mode's calculations. Each works with the other.

The VTScada Function Reference has the following to say about GetLog's Mode parameter:

Mode: Required. Indicates the mode of data collection.

Note that the mode is useful only when the TPP(*) parameter is valid and greater than 0. Mode may be one of:

Mode	Data Collection
0	Time-weighted average
1	Minimum in range
2	Maximum in range
3	Change in value over the range
4	Value at start of range
5	Time of minimum in range
6	Time of maximum in range
7	Sum of zero to non-zero transitions
8	Sum of non-zero time
9	Totalizer

10 Interpolated

11 Difference between the start and end values of a range (see comments in GetLog)

It is possible to retrieve more than one mode in a single GetLog statement. To do this, pass an array of values in as the Mode parameter.
(*) "TPP" in the above description, is the TimeRange parameter. In the QuarterHour Snapshot example, this was set to 900 seconds (15 minutes) and used with a mode of 4 (value at start of range).
By adjusting these two parameters, and using the example code shown in the topic "Common Features of a Report Module," you can create a wide variety of reports. The following table provides a few suggestions:

Mode	Time Range (TPP)	Report
7	3600 (1 hour)	Pump starts per hour
8	86400 (1 day)	Daily total running time
9	3600	Hourly totals
1 & 2 in an array	900	Minimum and Maximum values each quarter hour

Related Functions:

... GetLog

A 15-Minute Snapshot Report

This example shows how to create a new type of report, and how to add a new module to an existing VTS program. The result will be a snapshot report that works on a fifteen-minute basis rather than hourly or daily.

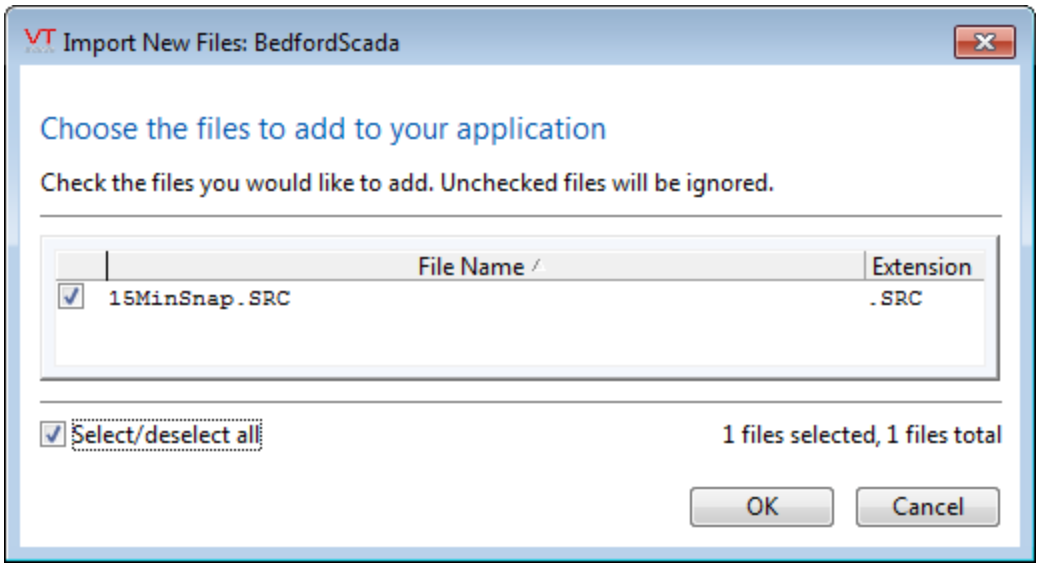
1. Select an existing application, or create a new one.
   Do not select or create a script application.
   Do not risk disaster by experimenting within a running production application.
2. Using a text editor, create a new file in that application's folder.
3. Name the file "15MinSnap.SRC".

4. Copy the code following step 10 into that file and save it.
5. Using a text editor, open the application's AppRoot.SRC file.
6. Declare the module within the (PLUGINS) section.
   The result should appear as follows. Note that the filename is case sensitive – you must enter upper and lower case letters in the declaration, exactly as you named the file.

```
[ (PLUGINS) {===== Modules added to other base system modules =====}
   15MinSnap Module "15MinSnap.SRC";
]
```

   (There will already be a (PLUGINS) section – do not add a second one.

7. Save the file and click the application's Import File Changes button.



8. Click OK to import the new module.
9. Start the application if it is not already running. (It was not necessary to stop it to do the preceding steps.)
10. Open the Reports page. Your new report should be available in the list of report types.

```
{================================ 15MinReport
==============================}
{ This plugin modifies the hourly snapshot report to be every 15
minutes      }
{ Groups : Loggers
         }
{ Areas : All
         }

{=================================================================-
=========}
```

```
(
   Reporter { Object value for call-backs };
   Start { Starting time };
   End { Ending time };
   Tags { List of tag names to report on };
   Vars { List of vars within tags };
)
[
  { Set up this module to become a plug-in for the reports }
   [(POINTS)
      Shared Report;
   ]

   Constant TypeFilter = "Loggers" {type of tags to use in the
report};
   Constant ReportName = "15 Minute Snap" {title for the report };
   TimeStamp { Time of last value returned };
   Obj { Instance of report };
]

Init [
   If 1 Wait;
   [                                          { 15 minutes = 900
seconds }
      Obj = \SnapshotReport(Reporter, Start, End, Tags, Vars, 900,
ReportName, 4 );
   ]
]
Wait [
   TimeStamp = Obj\TimeStamp; {ensures that the report object was cre-
ated before this module ends }
   If !Valid(Obj);
   [
      Slay(Self, 0);
   ]
]
```

Troubleshooting:

- The application won't compile.

  There is a typographic error in your code. Note the line number given in the error dialog. This gives you a starting point for locating the error.

- The report is not available.

  Ensure that you typed the code exactly as shown.

  Ensure that the declaration was placed in the existing (PLUGINS) section of AppRoot.SRC, and was placed before the closing square bracket of that section.

  Ensure that the Load File Changes button was pressed and no error dialogs opened as a result.

Related Information:

...Build Custom Reports - Discussion and instructions for creating custom reports

Diagnostic Files

In some VTScada applications, you may decided that there is a requirement for diagnostic files (as an example, a SQL data logger diagnostic file in which all queries to the SQL server are written for later analysis). The following list contains guidelines and tips on diagnostic files and their storage.

- Do not create diagnostic files in your application directories. The size limitation for synchronization of files by the RPC Manager is 32 MB. If a diagnostic file should grow beyond 32 MB in size (which can easily occur) your application will hang as it attempts to synchronize it. Additionally, once a diagnostic file has been locked by VTScada (when an "Update All" remote configuration action is performed), it can no longer be written to by VTScada.
- The file synchronization dialog shows the name of the file that it has just completed, rather than the name of the file that it is attempting to transfer.
- When file synchronization locks up for an apparently unknown reason, look at the files in your application directory and sort them by size. Any files greater than 32 MB are likely the culprit.
- As a minimum, always add a "\" character to the names of the files created for diagnostics (or for any other purpose) to prevent them from being created in your application directory. A better way to do this is to define a diagnostics directory somewhere to dump these files. (The configuration variables is a good place for creating path variables for this purpose.) Another way is to create the diagnostics file with a .LOG or .DAT extension (rather than with a .txt extension), as VTScada will not automatically add these files to the application during synchronization.

Related information that you may need:

...File I/O

...RPC Manager Functions

Working with Speech

The following VTScada modules allow you to provide speech features in your applications:

- Configure: Enables you to define how a speech stream will sound, and where it will be heard.
- GetDevices: Runs in the VoiceTalk thread and returns a list of devices available on a SAPI text-to-speech stream.
- GetVoices: Runs in the VoiceTalk thread and returns a list of voices available on a SAPI text-to-speech stream.
- Reset: Stops a speech stream and cancels any buffered speech.
- ShowLexicon: Displays a SAPI text-to-speech engine lexicon dialog to permit modification of pronunciation.
- Speak: Executes on the speech thread to speak supplied text through a specified SAPI text-to-speech stream.
- VoiceTalk: Opens and returns a handle to a SAPI text-to-speech stream.

Note that the following modules have been obsolete for some time, but are still provided for backward compatibility:
SpeechStream, SpeechEnum, SpeechLexiconDlg, SpeechReset, SpeechSpeak, and SpeechSelect.

Related Functions:

... Configure

... GetDevices

... GetVoices

... Reset

... ShowLexicon

... Speak

... VoiceTalk

Interrupt the Shutdown Process

There are two methods to interrupt the shutdown process. Note that neither can be used to delay a shutdown caused by time-limited trial license by more than ten minutes.

Method 1:
Any module declared in AppRoot.SRC as a member of the class (SHUTDOWN_HOOK) will run automatically during the shutdown process. Such a module may be used to write extra information to disk or perform any other task before the shutdown process completes.

Method 2:
If you add a module named "VAMStopAppCheck" to your application, you can interrupt the shutdown process to prompt for confirmation or to give the operator time to perform some task before proceeding with the shut-down. For example, when the TraceViewer is shut down, it will check whether logging is still enabled and if so, ask the operator whether to continue logging.

Note: This does not apply to shutdowns initiated by a low UPS. Those are considered to be both time-sensitive and critical and therefore will not be delayed by a VAMStopAppCheck module.

If adding this module to a script application, define it in the AppRoot.src file of the application. If adding it to a VTScada application, the module must be in it's own file, which is declared in the AppRoot.SRC file. (Note – it should be declared on its own, not within any of the module classes.) It is sufficient that the module be declared in the AppRoot.SRC file in order for the VAM to call it upon shut-down. This will happen when the user tries to stop the application by using the stop button in the VAM or by stopping the VAM itself. The VAM will not automatically call VAMStopAppCheck when the user closes the application by clicking the X in the corner of the title bar. Script applications are able to trap for this method of closing the application, but VTScada applications are not.

VAMStopAppCheck should be declared with two parameters:

- OKStopPtr – a pointer to be set. When *OKStopPtr is set to 1, the application may stop.
- VTSExit – a Boolean. If TRUE, VTScada is being shutdown. If FALSE, only the application is being stopped.

The general structure of the module will display a dialog to the operator, and wait for a response before continuing.

Example:

```
<
{============================== VAMStopAppCheck
========================}
{ Module called by VAM when Stop button is pressed. Setting OKStopPtr
}
{ to 0 tells the VAM not to stop the app. Setting it to 1 allows it
to  }
{ stop.                                                                 }
{=======================================================================-
===}
VAMStopAppCheck
(
  OKStopPtr             { Pointer to set: 1 if ok to stop, 0 if not
ok  };
  VTSExit               { Flag - TRUE if VTScada wants to exit; false
if app
                          is just being stopped. Default is false. };
)
[
  Close                 { TRUE if user chooses to stop the app
};
  CloseDialog Module    { Presents dialog to user getting con-
firmation
                          for app stop.                           };
]

Check [
  If 1 Wait;
  [
     { User has attempted to stop the application. Show the close dia-
log. }
    CloseDialog(&Close);
  ]
]

Wait [
  If Valid(Close);
  [
    IfElse(Close,
      *OkStopPtr = 1;
    { Else }
      *OkStopPtr = 0;
    );
```

```
    Slay();   {  In either case, this module is finished. }
  ]
]

>

<
{============================== CloseDialog
=========================}
{ Launched module presents a dialog to the user when the application
}
{ is stopped
}
{=========================================================================-
==}
CloseDialog
(
  AskExitResultPtr            { Pointer to set the result
};
)
[
  AskExitResult               { The user's choice
};
]

CloseDialog [
  AskExitResult = \System\4BtnDialog(\System\Question_Icon,
                                     "Yes", "No", Invalid, Invalid,
                                     "Continue shutdown?", Invalid,
Invalid,
                                     1, 0, 1, "Continue with shut-
down?",
                                     Invalid, Invalid, Invalid, 2,
                                     Invalid, Invalid);
  If Valid(AskExitResult);
  [
     *AskExitResultPtr = AskExitResult == 1 ? 1 : 0;
     Slay();
  ]
]
>
```

Alarm Manager

The Alarm Manager maintains a record of alarm activity, and keeps track of the current status of all alarms in the system, whether they are active, acknowledged, shelved, etc. This information is stored as records in an alarm database.
To record and store information, the Alarm Manager uses a VTScada Historian. By default, the System Alarm Historian is used, but others may be created and selected as part of application development.

Note: Alarm data and process I/O data should always be stored with separate Historians.

Alarms are linked to Historians via Alarm Database tags. Two Alarm DB tags are standard with every VTScada installation: System Alarm DB is the default for all user-created alarms and events. System Event DB is used for all built-in VTScada events including security logs and operator-control actions. If you create new Alarm Database tags, only (and all) those alarms that are children of a database tag will be stored in that database. In all other cases, the default is the System Alarm DB.
Note that the alarm files, Alarms.DB and Alarms.LOG are obsolete as of VTScada version 11.2. For legacy applications that are upgraded to 11.2 or later, all alarm history will be transferred to Historians as a one-time process. In the event that you intend to add extra Alarm Database tags to your application and store certain alarm information with those custom databases, do so before transferring the alarm history.

Programmers can use the Alarm Manager API described in this chapter for the following tasks.

- Add alarm features to tags that they code from scratch.
- Customize the columns and other display characteristics of alarm lists.
- Query alarm status information for use in modules such as custom reports.
- Query or modify alarm properties to view or change the Alarm Manager configuration.

Related Information:

...Adding Alarms to Custom Tags

...Alarm Functions

...Alarm API Structure Definitions

...Alarm Manager Function Constants

...VTScada Event Logging

...Alarm Message Templates

See the VTScada Admin Guide for:

...Application Properties for Alarms

...Properties for the Alarm Notification System

Alarm API Structure Definitions

The alarm manager defines several data structures.

- The configuration structure is obtained and populated when commissioning an alarm. Alarm Configuration Structure
- The status structure may be obtained and monitored to watch for changes to an alarm's state. Alarm Status Structure
- The transaction structure contains information about an alarm event that is stored to from the database including what changed (active state on or off...), the logged-on user, which workstation, etc. Alarm Transaction Structure
- The record structure is the complete set of information for each entry in the alarm database. This includes both configuration and transaction information. Alarm Record Structure

Alarm Configuration Structure

Every alarm can be described using a known configuration structure. When configuring a new alarm, this structure should be created via a call

to \AlarmManager\GetAlarmConfiguration, then populated with the appropriate values before a call is made to \AlarmManager\Commission.

ConfigurationStruct { All Boolean flags default to FALSE }	
Name	Unique name for the alarm
FriendlyName	Display name of the alarm's source
Area	Area
Description	Description. Was "Message" prior to 11.2
Priority	Priority. Must be valid to be commissioned. Must be an integer corresponding to the Alarm Priority tag values.
Reserved	
Disable	TRUE to disable the alarm
DisableParmName	Name of the tag's disable parm. Allows us to get the operator name who made the config change.
OnDelay	Seconds to delay before activating
OffDelay	Seconds to delay before clearing
RearmDelay	Seconds to delay before rearming after ack
Setpoint	Setpoint of alarm evaluation
ValueLabels	Array of labels to display instead of Value or Setpoint. Rarely used by tags other than digitals.
Units	Setpoint units
Function	Enumerated function for alarm evaluation (1)
AlarmType	String identifying the type of alarm
Trip	TRUE if alarm only becomes unacked not active
NormalTrip	TRUE if alarm becomes unacked when it clears
OffNormal	TRUE if alarm only becomes active not unacked
Deadband	Setpoint deadband
PopupEnable	TRUE to enable popup display of active alarm
SoundFile	Filename relative to app path of custom sound
Custom	Array/Dictionary/Structure of custom fields
AdHoc	TRUE if alarm is ad hoc

GetAlarmConfiguration returns only a copy of the alarm's structure, not a reference. To update any property within the structure:

1. Obtain a copy using GetAlarmConfiguration.
2. Change values within that copy as required.
3. Include the copy in a new call to the Commission function.

Related Functions:

...GetAlarmConfiguration

...Commission

Alarm Status Structure

AlarmStatus Struct	
IsActive	TRUE if alarm is on the Active list };
IsUnacked	TRUE if alarm is on the Unacked list };
IsShelved	TRUE if alarm is on the Shelved list };
IsDisabled	TRUE if alarm is on the Disabled list };

This structure should be used in all new code, replacing the older functions IsActive, IsUnacked, IsShelved and IsDisabled.

Related Functions:

...GetAlarmStatus

Alarm Transaction Structure

[

TransactionInfo Struct	
Name	Alarm name
Cfg	Alarm's configuration. Only required to bypass the alarm's commissioned configuration.
Action	Alarm Action code
Transaction	Alarm transaction string
Timestamp	Alarm timestamp (UTC)

TransactionInfo Struct	
Value	Tag value
Custom	Custom fields; overrides Cfg\Custom
Reserved	
MachineID	Workstation's MachineID. Defaults to local.
AccountID	AccountID of operator
Device	Name of client device
ExpiryTime	Time the record is to be removed (UTC)
RemovalGUID	Reference GUID for targeted removals
NoteInfo	Structure containing information about a note that has been attached to this transaction. It provides the Timestamp and GUID of this transaction.

The transaction string takes the form, "ListName+" or "ListName-". Several transactions can be combined in one string. The following example sets both the unacknowledged status and the active status off: “Unacked-Active-”.
List names include the following: Active, Unacked, Shelved, Disabled, Configured.

Alarm Record Structure

AlarmRecord Struct	
TimeStamp	UTC time for the event
GenTimestamp	UTC time when the record was written
GUID	16 byte unique ID for the event
ReferenceGUID	GUID of original event that this cancels
ReferenceTime	Time when original canceled event occurred
Name	Alarm name, typically the tag name
Area	Alarm area
Transaction	List additions and deletions
Action	Alarm action to display for history event
Priority	Alarm priority (integer)

AlarmRecord Struct	
IsShelved	TRUE if this alarm is shelved
Database	UniqueID of the alarm database tag
Custom	Field to be used by OEM & app code
MachineID	Workstation where record originated
Device	Name of originating client computer
UserID	User associated with the event
Description	Alarm description/message
Reserved	
ExpiryTime	Time shelved record is to be removed
OnDelay	Time to wait (seconds) before activating
OffDelay	Time to wait (seconds) before clearing
Value	Tag value at time of alarm event
Setpoint	Alarm setpoint
Units	Setpoint units
Function	Setpoint function
AlarmType	Type of alarm
SetpointLabel	Label to display instead of Setpoint
ValueLabel	Label to display instead of Value
RearmDelay	Rearm delay (seconds)
Deadband	Analog deadband
SeqNum	Sequence Number (for history sort)
Trip	TRUE if alarm was tripped
AdHoc	TRUE if alarm was ad hoc
NoteAttached	TRUE if one of more notes attached to record
IsConfig	TRUE if the alarm configuration is changed

Alarm Manager Function Constants

The Alarm Manager defines the following constants. Use these in custom code that enables the user to choose the trigger for an alarm condition.

ALM_FUNC_ON_CHANGE	upon change of value
ALM_FUNC_EQUAL	==
ALM_FUNC_NOT_EQUAL	!=
ALM_FUNC_LESS_THAN	<
ALM_FUNC_LESS_EQUAL	<=
ALM_FUNC_GREATER_THAN	>
ALM_FUNC_GREATER_EQUAL	>=
ALM_FUNC_AND_WITH	&&
ALM_FUNC_OR_WITH	||
ALM_FUNC_XOR_WITH	^
ALM_FUNC_NOT_AND_WITH	!(&&)
ALM_FUNC_NOT_OR_WITH	!(||)

VTScada Event Logging

An alarm may be described as "A situation to which an operator must respond", while an event is "an action that is recorded, but requires no response." Events differ from alarms only in purpose and notification. For both, a triggering action occurs, and a transaction is recorded of that occurrence.

Many events are built into VTScada. Each change to security con-figuration is logged as an event recording who, what, when and where (workstation). Operator logons and log-outs are similarly recorded. Each operator-control action is recorded as is each VTScada action such as sending an alarm notification, generating a scheduled report from a

Report tag, and more. All of these VTScada-generated events are logged in the Alarm Event DB tag.
All events can be viewed in the History page of any Alarm List.

Logging of Operator Control Actions

Each time an operator performs a control action in an application, such as starting a pump, that action is logged. The record includes the operator name, the workstation used, a timestamp, the name of the output tag, the area of the output tag, the current value, and the value written. If no tag area property is defined or available, then the value defined in the property, OperatorLogArea will be used.
The content of the logged message is controlled by the application property OperatorLogTemplate. Prior to version 11.2, messages longer than 80 characters were trimmed, but this is no longer the case. Other properties (see list of related information) control whether operator logging is enabled and provide default values to be used when none are otherwise available.

Logging of Security Events

Each time a user logs on or logs off, or a manager performs a security-related action such as adding a new user account, details about that action are added to the alarm log.
For each security entry in the History list, the following information is displayed:

- The name of the workstation on which the security event occurred,
- The account name of the person who performed the action
- A time and date stamp,
- Details about the action performed

Related Information:

Refer to the VTScada Admin Guide for:

...OperatorLogArea

...OperatorLogging

...OperatorLogName

...OperatorLogTemplate

Query the Alarm History

The History tab of the Alarm Page (or any Alarm List) offers the easiest way to query alarm history with its various filters. Note that you can use the keyboard combination CTRL+C to copy the information from a report (or any alarm list), then paste it into a spreadsheet for further processing.

In custom code, you can build your own alarm lists by using the function GetAlarmList. This is the function used by the Alarm Page and every Alarm List widget.

You can also query historical alarm data using code via the SQL Interface command, "SQLQuery". You may use this function to build SQL statements that can select data directly from the list of active alarms or the alarm history.
Two tables are available for you to query. :Alarms for current alarms and :AlarmHistory for past events. Note the leading colon in both table names.
For example, to query the names and priorities of all active alarms:

```
SQL_Query = "SELECT Name, Priority FROM :Alarms WHERE Active = 1";
\VTSSQLInterface\SQLQuery(SQL_Query, &Result, &FieldNames,
&FieldTypes,
                          &RetCode, &ErrorMessage);
```

Upon success, the resulting two-dimensional array will be stored in the variable Result. The size of the first dimension is controlled by the number of fields you query for. The size of the second dimension matches the number of rows returned.
In the event of an unsuccessful query, Invalids will be returned in the SQLQuery parameters.

To retrieve all events associated with the operator Bob on April 10, 2008 :

```
SQL_Query = Concat("SELECT  Timestamp, Name, SubName, Event ",
                   "FROM :AlarmHistory ",
                   "WHERE  Timestamp >= '2008-04-10 0:00:00' ",
                   "AND Timestamp < '2008-04-11 0:00:00' ",
                   "AND Operator = 'Bob'");
\VTSSQLInterface\SQLQuery(SQL_Query, &Result, &FieldNames,
&FieldTypes,
                          &RetCode, &ErrorMessage);
```

To know when the query has finished, watch for RetCode becoming valid. Available column names to use in your query include:

Timestamp	Name	SubName
Event	Message	Priority
Type	HookPointValue	Area
HookPointUnits	Operator	

Related Information:

...Alarm Reports - Using the built-in alarm reports

...GetAlarmList - VTScada function reference

...SQLQuery - VTScada function reference

...SQL Queries of VTScada Data: The ODBC Server - VTScada Developer's Guide - Configuration and examples.

Alarm Message Templates

Templates can be used to define the alarm that is delivered via the Alarm Notification System (voice, email or pager) or via the text-to-speech feature. These templates can use a combination of words and replaceable symbols to define the content of the alarm message. Templates may be up to 128 characters in length.

Related Information:

Referring to the VTScada Admin Guide, separate templates are available for each of:

- Spoken alarms
    - AlarmSpeechTemplate
- Dialed voice messages
    - AlarmDialerTemplate
    - AlarmDialerStatusTemplate
- Emailed alarms
    - AlarmEmailTemplate
    - AlarmEmailAckSubjectTemplate
    - AlarmEmailAckTemplate
    - AlarmEmailStatusTemplate
    - AlarmEmailSubjectTemplate
- Paged alarm messages
    - AlarmPagerTemplate
    - AlarmPagerStatusTemplate
- SMS messages
    - AlarmSMSTemplate
    - AlarmSMSStatusTemplate
    - AlarmSMSAckTemplate

There are also two template configuration options that set the format used for the date and time if these parameters are used as part of the template.

- Date Format see: AlarmTemplateDateFmt
- Time Format see: AlarmTemplateTimeFmt

The complete list of replaceable parameters:

Parameter	Meaning
%A	Area of the Alarm tag.
%D	Date of the alarm
%F	Full tag name

%H	Short tag name
%M	Alarm description
%N	New sentence for email and pager messages.
%O	Name of the operator logged on at the time the alarm was triggered.
%P	Priority of the alarm.
%S	Status of the alarm
%T	Time of the alarm
%U	Units of the Triggering tag.
%V	Alarm value (this is the value of the alarm trigger at the time that it triggered the alarm)
%W	Pause for ¼ second. Has no effect on email or pager messages.

Custom Alarm Hook API

In versions of VTScada prior to release 11.2, developers who wanted to add custom functionality to an alarm event would override a module of the Alarm Manager to add their code. That technique is now obsolete. Many existing overrides will continue to work, but should be tested before being put into production use with version 11.2 or later.
To add extra functionality to an alarm transaction, create an alarm hook module. A set of hook names has been defined within the Alarm Manager. If your application contains a module with a matching name, it will be called just before the transaction is logged, allowing you to perform extra work.
An alarm hook should return TRUE or Invalid to allow the transaction to proceed and be logged. Alarm hooks that return FALSE will stop the transaction from proceeding.
The module may be defined with one parameter, which VTScada will use to pass in the fully-populated transaction structure.

Defined alarm hooks:

- AlarmAckHook
- AlarmActiveHook
- AlarmCommissionHook
- AlarmDecommissionHook
- AlarmDisableHook
- AlarmEnableHook
- AlarmEventHook
- AlarmModifyHook
- AlarmNormalHook
- AlarmNormalTripHook
- AlarmOffNormalHook
- AlarmPurgeHook
- AlarmRearmHook
- AlarmShelveHook
- AlarmTripHook
- AlarmUnshelveHook

Example 1 Do something extra when the alarm closes:

```
<
AlarmNormalHook
(
 { parameter not required }
)
[
 { ... local variables ... }
]
Main
[
  If 1;
  [
     { Do something like write out a value to a PLC }
    DoSomething();
     { Returning Invalid or TRUE allows the AlarmManager to log the
Transaction when we're done }
    Return(TRUE);
  ]
]
>
```

Example 2: Clear the alarm when it is acknowledged:

```
<
AlarmAckHook
(
  TransactionStruct { transaction structure };
)
Main
[
  If 1;
  [
     { Acknowledge and Clear the alarm }
    TransactionStruct\Transaction = Concat(Trans-
actionStruct\Transaction, "Active-");
     { The additional transaction text is added to any that already
exist. }
    Return(TRUE);
  ]
]
>
```

Related Information:

...Alarm API Structure Definitions - Includes the transaction structure.

Customize Columns in Alarm Displays

The structure of lists in the Alarm Page and in Alarm List Widgets is controlled by the XML file, C:\VTScada\VTS\AlarmListFormats.XML. You may decide to customize the structure for any of the following reasons:

- Add or remove columns in a list.
- Set the default width of columns.
- Add customized columns to a list.
- Add a customized list format.

Note: Do not edit C:\VTScada\VTS\AlarmListFormats.XML. Your changes will be lost with your next VTScada update.
Do not copy this file to your application folder. Local definitions of column formats and list formats that you have not customized will prevent updates from taking effect.

To make the customizations described in this topic, create a file named AlarmListFormats.XML in your application folder. The structure must be as described in this topic. You may copy sections of

\VTScada\VTS\AlarmListFormats.XML to use as a template, but do not save any definition that you do not intend to customize. Your custom definitions will override or be added to those from the VTScada file. The structure of the file is as follows:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
  <AlarmList>
    <ColumnFormats>
      <Format name="SingleSet1">
        <Column width="160">AlarmCellTimestamp</Column>
        ... more column definitions ...
      </Format>
      ... more format definitions ...
    </ColumnFormats>

  <ListFormats>
    <Format name="AlarmStandard" label="AL_StandardFormatLabel">
      <list name="History" label="AL_HistoryListLabel">
        <Single>SingleSet1</Single>
        <Double>DoubleSet1</Double>
      </list>
      ... more list definitions...
    </Format>
    ... more format definitions...
  </ListFormats>
</AlarmList>
```

XML Format Hierarchy:

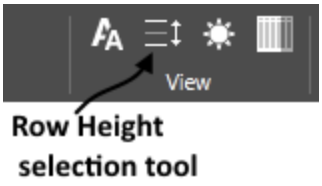
- ListFormat definitions are linked to user-interface tools, and are selected according to rules coded into that tool. Examples follow.
- Within each ListFormat will be one or more lists such as Active, Current, etc.
- Each list definition within each ListFormat will contain two versions, Single and Double. This selection is controlled by the operator by toggling the Row Height option.
- The single version and the double version each specify a ColumnFormat.
- Two ListsFormats may each contain a list with the same name such as "History", but these are separate definitions. Different column formats can be specified for History (and any other list) in different ListFormats.
- ColumnFormats specify the display modules to be shown in the column cells, the order of the columns from left to right and the default width of each column.

For example, an alarm popup uses the "PopupStandard" list format, which contains only one list: "Unacked". In the Alarm Page, the default is

to show the AlarmStandard format, containing History, Active, Unacked, etc, but if an operator chooses to view only the System Event DB, then the "EventStandard" list format will be selected automatically limiting the selection to just the History list.
Application properties are used to set the text used for the labels of lists and columns in the user interface. For example:
AL_HistoryListLabel = History

Column Format Definitions:

The set of lists shown in the Alarm Page is predefined, but you may alter the appearance of any list.
For each list, History, Active, etc. two sets of column formats are defined. Two are required so that the operator may use the Row Height button to switch between a list with one item per column and a list with (in some cases) two.



Standard display

Time			Ack	Status	Area	Name	Description	Value	Setpoint	Units
2015-12-21 10:56:29	2		Ack	Alarm	Zone 1	Local TCP Port\..\Tank Level	Monitor tank level HIGH	90	90	%
2015-12-21 10:55:33	1		Ack	Normal	Zone 1	Local TCP Port\...\Critical Level	Water level critically high	90	95	%

Columns stacked after using the Row Height button

Date Time		Status Ack	Area	Name Description	Value	Setpoint
2015-12-21 10:56:29	2	Alarm Ack	Zone 1	Local TCP Port\PLCSim\Tank 1\Tank Level Monitor tank level HIGH	90 %	90 %
2015-12-21 10:55:33	1	Normal Ack	Zone 1	Local TCP Port\PLCSim\Tank 1\Tank Level\Critical Level Water level critically high	90 %	95 %

In the ColumnFormats section of the XML file, these are given generic names. The display name is set in the second section of the file. An example of the format follows:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<AlarmList>
  <ColumnFormats>
    <Format name="SingleSet1">
      <Column width="160">AlarmCellX</Column>
```

```
    <Column width="26">AlarmCellY</Column>
    ...
  </Format>
  <Format name="DoubleSet1">
   <Column width="26">AlarmCellX</Column>
   ...
```

Each "<Format" section sets the columns to be included. Columns are displayed in the list from left to right in the order found in the Format section.

The width sets the default size to be used by that column. Changes to column widths by the operator are stored on a per-operator, per-session basis, overriding your defaults. There are three ways to specify the width of a column:

- Column width = "30". Sets the specific number of pixels to be used by the column.
- Column width = 30%. This column will occupy 30% of the area remaining after the columns with specific width settings have been accounted for. The total of percentages should be 100 or less. See next option.
- No width specification. All columns with no width specification will share equally the space remaining after columns with a specific or percentage width have been accounted for.

The example text, "AlarmCellX", "AlarmCellY", etc. must be replaced, either by one of the VTScada modules provided to format and display the contents of each cell in an alarm list, or by a module of your own creation. See link under Related Information.

Example:

For the standard alarm page display of unacknowledged alarms, move the name and description to the first column.

This will require a change of the ColumnFormat ordering, but the first step is to discover which ColumnFormat is used by the Unacked list in a standard display. Fortunately, the names in the XML file make this easy to find:

Under <ListFormats> find <Format name="AlarmStandard" ...> You can safely assume that this is the standard format. Within that section, find the list named "Unacked":

```
<list name="Unacked" label="AL_UnackedListLabel">
  <Single>SingleSet2</Single>
  <Double>DoubleSet2</Double>
</list>
```

From the above, it is clear that ColumnFormats SingleSet2 and DoubleSet2 are used. These can be copied from the original file, together with enclosing XML specifiers and reordered. The file you save to your application as "AlarmListFormats.XML" should look like the following. Only the two column formats are being overridden in your application; all others will continue to use the default XML file. Don't forget to import file changes to add your version of the XML file to the application.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<AlarmList>
  <ColumnFormats>
    <Format name="SingleSet2">
      <Column>AlarmCellName</Column>
      <Column>AlarmCellDescription</Column>
      <Column width="26">AlarmCellPriority</Column>
      <Column width="160">AlarmCellTimestamp</Column>
      <Column width="50">AlarmCellAck</Column>
      <Column width="18">AlarmCellIcon</Column>
      <Column width="80">AlarmCellAction</Column>
      <Column width="140">AlarmCellArea</Column>
      <Column width="90" extra="1">AlarmCellValue</Column>
      <Column width="90" extra="1">AlarmCellSetpoint</Column>
      <Column width="70" extra="1">AlarmCellUnits</Column>
      <Column width="18">AlarmCellNote</Column>
    </Format>
    <Format name="DoubleSet2">
      <Column>AlarmCellDoubleNameDescription</Column>
      <Column width="26">AlarmCellDoublePriority</Column>
      <Column width="83">AlarmCellDoubleTimestamp</Column>
      <Column width="18">AlarmCellIcon</Column>
      <Column width="80">AlarmCellDoubleActionAck</Column>
      <Column width="140">AlarmCellArea</Column>
      <Column width="90" extra="1">AlarmCellDoubleValue</Column>
      <Column width="90" extra="1">AlarmCellDoubleSetpoint</Column>
      <Column width="18">AlarmCellNote</Column>
    </Format>
  </ColumnFormats>
</AlarmList>
```

The "extra" attribute, when present and set to 1, enables the visibility of the column to be toggled by the Show/Hide Extra Columns tool.
There is also an “alwaysShowShelved” attribute. For example:

```
<list name="Shelved" label="AL_ShelvedListLabel" alwaysShowShelved-
d="1">
```

This attribute, when true, enables records that are marked as “shelved” to appear in the list even when the “Show Shelved Alarms” tool is not toggled.

Related Information:
Alarm Column Graphics Modules

Alarm Column Graphics Modules

The following are the names of graphics modules provided by VTScada to display information in Alarm Lists. The file, AlarmListFormats.XML controls which module is used by each column of each alarm display list. "Double" in a module name signifies that it is designed for use when operators click the Row Height button to switch columns from a single item to doubled items.
AlarmCellAck
AlarmCellAction
AlarmCellAlarmType
AlarmCellArea
AlarmCellDeadband
AlarmCellDescription
AlarmCellDevice
AlarmCellDisabledIcon
AlarmCellDoubleActionAck
AlarmCellDoubleDeadband
AlarmCellDoubleDescription
AlarmCellDoubleExpiryTime
AlarmCellDoubleIconPriority
AlarmCellDoubleNameDescription
AlarmCellDoublePriority
AlarmCellDoubleSetpoint
AlarmCellDoubleTimestamp
AlarmCellDoubleValue
AlarmCellDoubleWorkstationDevice
AlarmCellExpiryTime
AlarmCellIcon

AlarmCellName
AlarmCellNote
AlarmCellOffDelay
AlarmCellOnDelay
AlarmCellOperator
AlarmCellPriority
AlarmCellPriorityColor
AlarmCellPriorityText
AlarmCellRearmDelay
AlarmCellSetpoint
AlarmCellShelvedIcon
AlarmCellTimestamp
AlarmCellUnits
AlarmCellValue
AlarmCellWorkstation

You can override any of these modules with your own versions. The overall structure should be similar to the following example. The content is up to you.

```
AlarmCellArea(Parms)
[
  Title = "AL_AreaColumnLabel";
  SortKey = "Area";
]
Main [
  GUIText(…, Parms\TextColor, Parms\Font, Parms\Area);
  WinTooltipCtrl(…);
]
```

The Parms parameter is a VTSCada-supplied structure containing the elements shown in the following table. Information about each alarm that is displayed in the cell, and the defaults for how it is to be displayed are both passed to the module using this structure.
AckedIcon
AckedIconWd
ActiveIcon
ActiveIconWd
Action

AlarmType
Area
ConfiguredIcon
ConfiguredIconWd
Custom
Deadband
Description
Device
DisabledIcon
DisabledIconWd
DrawAlarmList
ExpiryTime
Font
FriendlyName
Function
GreyTextColor
GUID
HasNote
HistoryIcon
HistoryIconWd
IsActive
IsDisabled
IsHistory
IsShelved
IsUnacked
MessageID
Name
NoteIcon
NoteIconWd
OffDelay
OnDelay
Popup
PriorityColor
PriorityEvent

Priority
PriorityText
PriorityTextColor
RearmDelay
Record
RecNum
RelativeName
Root
RowColor
Session
Setpoint
SetpointLabel
ShelvedIcon
ShelvedIconWd
SmallFont
TagDescription
TextColor
Timestamp
Units
UserID
UserName
ValueLabel
Value
Workstation

Configuration Management

Configuration Management is a term that describes VTScada features related to all configuration changes.
This includes ChangeSet files, the ability to read and write configuration files, the ability to make inquiries about the current Layer object (application) and the Version Control system.
Features of the Configuration Management System:

- Does not depend on a configuration server.
- Collaboration not limited by file locking.
- Not bound to the RPC network
- Provides a complete audit trail.
- Easy to detect off-line changes since these must be merged into the system by an authorized user.

For any application, you can discover who has the working copy lock by selecting that application in the VAM and pressing the keyboard combination, "Ctrl-L".

Related Information:

...Configuration Management API - Reference for functions that make up the Configuration Management system.

Configuration Management API

The functions that make up the Configuration Management API are as follows.

Note: These functions should be used only by advanced VTScada programmers. Errors in the use of these functions can cause irreparable damage to your application.

Related Functions:

Most of the following functions are called against a Layer object(e.g. LayerRoot\Function()). The Layer object can be acquired using GetAppInstance, GetLoadedAppInstance or GetOEMLayer.

...AcquireLock - Subroutine to acquire an exclusive lock on reading/writing working copy files across all applications.

...AppIsRunning - Reports whether the application has been started and the start-up process is complete.

...AppIsStarted - Returns TRUE if the application has been started.

...AppIsStarting - Returns TRUE if the application is in the process of starting.

...ApplyChangeSetFile - Apply a named ChangeSet to an application layer.

...CaptureSettings - Gathers a single property value or an accumulated section and returns the result in a tabular format.

...Combine - Performs a Merge2 operation with automated conflict resolution and change priority.

...CommitEditedFiles - This function compiles and commits edited files if the compile succeeds.

...DirectApply - Applies a set of changes directly to the repository, without disturbing existing (non-conflicting) changes already on either branch.

...EditFile - Informs the configuration management system that a file has been modified in the working copy, typically before making a call to CommitEditedFiles.

...GetAppInstance - Asynchronously, retrieves the Layer object (LayerRoot) for a particular application specified by its GUID.

...GetCodeObj - Retrieves the "Code" object associated with the layer.

... GetINIProperty - Given an array of INIProperty structures, returns the value of a given property from that array.

...GetLoadedAppInstance - Synchronously, retrieves the Layer object (LayerRoot) for a particular application specified by its GUID.

...GetOEMLayer - Retrieves the layer root module of the OEM layer (should one exist) of the layer this is called against.

...GetPlatformInfo - Gathers information about the current application and the workstation it is running on.

...GetWCPath- Returns the full working copy path for an application.

...GetWCRevision - Returns the revision structure for the repository revision in use by the working copy.

...HasCompilationErrors - Reports if the working copy presently has unresolved compilation errors

...HasUndeployedChanges - Finds whether the local machine is maintaining changes that have not been deployed, including changes that have been recorded by EditFile but have yet to be committed.

...IsAppEditable - Returns TRUE if the application can accept changes without being re-started.

...IsOnLocalBranch - Returns TRUE if the local machine is maintaining changes that have not been deployed within the repository.

...IsRunOnly - Returns TRUE if the application is a run-file-only app, according to the WC contents.

...LayerInUse - Returns true if the application is running, or if there are any applications that depend on this layer, running or not.

...Merge - Applies a set of changes (the output of a Diff operation) to a buffer.

...Merge2 - Attempts to apply two different Diff buffers to a single origin buffer.

...ReleaseLock - Releases a working copy semaphore that was acquired by AcquireLock.

...ReadINIProperties - Gathers the sum of all of the properties files in this layer and all of its parents including the local workstation files.

... ReadPropertiesFile - Reads a single Settings file and returns an INIFile Structure.

...RecordProperty - Helper function used to record settings without needing to explicitly interact with the settings files.

...RepoSubscribe - Enables the caller to specify a callback which will be triggered whenever the application's repository changes.

... SetINIProperty - Given an INIFiles structure, this function sets the property with the specified name and section to the specified value

...Start - Start an application.

...LayerRoot\Stop - Stop an application.

...WriteINIProperties - Writes properties to the local layer's various settings files in one operation.

... WritePropertiesFile - Write a single Settings file according to the properties in an INIFile structure.

Communication Drivers

Communication drivers tend to become large blocks of code. This is due to the complexity of the protocols developed for many hardware devices. The VTScada side of the equation is usually quite straight-forward, but it often happens that a large number of subroutines will be required to handle all the details of a driver's protocol.
This chapter describes the software design process as it applies to the creation of communication drivers in VTScada. It is organized as follows:

- The fundamental concepts of how VTScada implements a communication driver are covered in the first few topics. If you have not written a driver before, start with Communication Driver Fundamentals.
- The information you should gather before beginning to write code is listed in the topic, Communication Driver Design.
- A step-by-step description of how to create a communication driver is provided, followed by detailed information on the mandatory and optional components of a driver. See: Writing a Communication Driver.
- A template for a simple driver is presented as an example. All hardware-specific code has been removed from this example, leaving only the VTScada components. See: Communication Driver Template.
- A reference section is provided, starting at The VTSDriver API, and including:
- The API of the built-in module, VTSDriver.
- Details on how driver information is distributed through a networked application.
- Rules for writing a driver.
- Information about tools for driver diagnostics and statistics gathering. See: Driver Diagnostic Tools.
- Instructions for installing and using a driver are given at the end of this chapter. If you have been given the code for a new driver, and are simply looking for the steps to add it to your application, you can skip ahead to the section: Install a New Driver (Example: GE 9030 SNP Driver).

Related Information:

Communication Driver Fundamentals

Summary:

- Communication drivers are tags. All the rules of tag structure must be followed when creating a driver.
- VTScada includes a standardized driver module, VTSDriver that provides a consistent interface to all I/O tags.
- VTSDriver relies on the communication driver tag to handle read/write requests according to the hardware's protocol.

VTScada collects information from industrial equipment through hardware input devices, displays the information graphically on a computer, and sends control signals back to hardware output devices to control the operation of the equipment. Computers and hardware input/output devices do not use the same protocol to communicate; therefore a data translator is required to provide communication between the devices. This is the role of a communication driver.

In VTScada, communication drivers are in the form of a specialized tag type. Because each type, make, and model of I/O device uses a different communication protocol, many different communication driver tag types are required to provide the interface from these different I/O devices to the VTScada software.

VTScada ships with communication drivers for the following I/O devices:

Allen-Bradley, CalAmp, CIP, Data Flow RTUs
DNP3, MDS Diagnostic, Modicon, Omron Host Link
OPC Client and Server, Siemens S7, SNMP

… and more.

Tag types can have unlimited instances, therefore many copies of the same or different drivers may run at the same time, providing communication between your PC and multiple I/O device drivers.

Related Information:

...Data Exchange between VTScada and a Driver

...What Happens Within the VTScada Code?

...Communication Driver Tags - See the VTScada Developer's Guide

Data Exchange between VTScada and a Driver

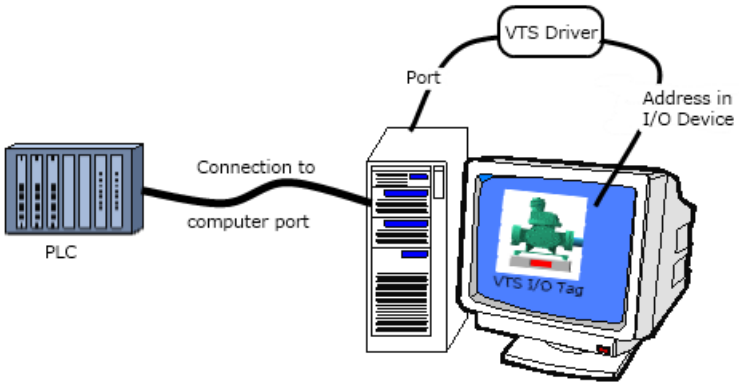
VTScada uses I/O tags that are assigned to read from or write to specific address in the PLC or RTU. These include analog input and output tags, digital input and output tags, status tags, control tags, etc.

All I/O tags have the following two properties in common:

- I/O Device: The "I/O Device" property tells the tag which driver tag it should use in order to communicate with the correct PLC or RTU.
- Address: The "Address" property tells the tag which memory location at the PLC or RTU to read from or write to.

All communication drivers have a common property:

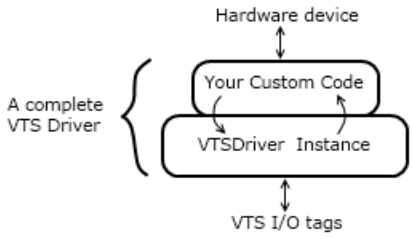
- Port: The "Port" property indicates to the device driver tag the correct serial port or TCP/IP socket that it should use to transmit and receive data from the PLC or RTU.



In summary, port tags provide communication over a physical connection between remote equipment and the PC. Communication drivers read and write data over that connection using the protocol required by the remote device. I/O tags within VTScada read data from or send data to the communication drivers in order to complete the link between the specific addresses in the remote equipment and the widgets on the screen of a VTScada application.

What Happens Within the VTScada Code?

The VTScada software module, "VTSDriver" (the source code for which is stored in a file named, "VTSDrvr.web"), provides the link between the communication driver and the VTScada application. A unique instance of VTSDriver is automatically created for every instance of every driver within an application.



Briefly, the role of VTSDriver is to:

- Distribute data to I/O tags
- Trigger polling

- Distribute data to client workstations via RPC.
- Synchronize data on startup.

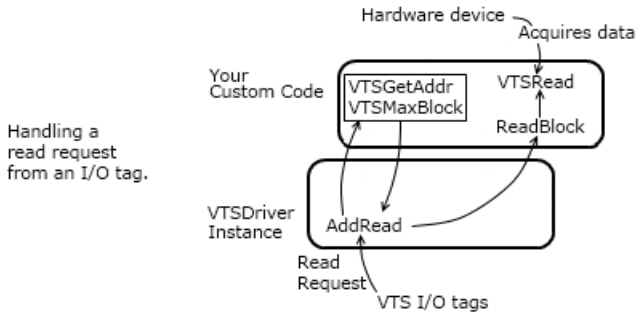
Reading Data

VTSDriver contains a module called, "AddRead" that is called by an input tag to create a request to read a specific range of memory. AddRead expects the following parameters:

- The address from the input tag.
- The number of elements (size of the block) to read.
- A pointer to a place to put the retrieved values.
- The rate at which to perform the read.

AddRead calls VTSGetAddr and VTSMaxBlock in the communication driver to get the information it needs to coalesce the addresses into appropriate blocks. A ReadBlock module is then launched, which determines the best organization of blocks and launches a separate VTSRead module for each actual block of data to be read.

VTSRead, VTSGetAddr and VTSMaxBlock all reside in the communication driver, not in the VTSDriver module. This is an important detail to note if you are planning to write your own communication driver.



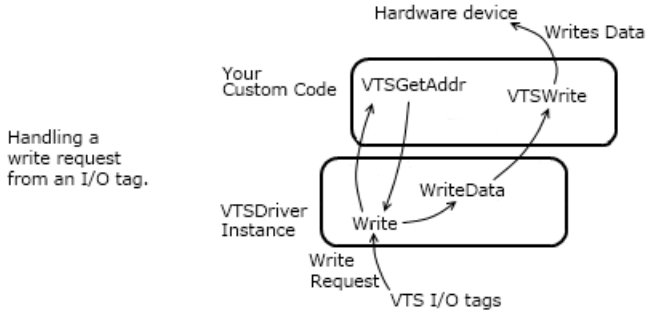
Writing Data

Whenever a value is changed in an output tag, a Write module in VTSDrvr.web is called. This module expects the following parameters:

- The address from the I/O tag.
- The number of elements to write.
- A pointer to the source of the written data.

- The data type.
- The name of the I/O tag that called this module to request a write.

The Write module launches WriteData, which in turn calls VTSGetAddr in order to obtain the information it needs to call VTSWrite. VTSWrite, a module in the communication driver, is then launched.



To emphasize, note that VTSWrite is a module that you create in the communication driver, not in VTSDrvr.web.

Rewriting Data

VTSDriver provides the ability to store and rewrite the last set of output values if requested or required. This feature is controlled by the following code that you may add to your custom driver:

Data Storage

Your custom driver must contain a variable named, StoreOutputs. When set to TRUE (1), VTSDrvr stores the last output value sent to each address. If later changed to FALSE (0), the stored values are erased so that they cannot be written accidentally.

Each VTScada Driver instance stores its own information on what value was written to what address in a retained dictionary. The dictionary is keyed on address and stores both the value and the tag name.

The information is populated only after a successful write has been carried out. It is stored via the module RememberOutputs().

Data Rewrite

A rewrite may be triggered either automatically, or manually.

A module in VTSDrvr named ForceRewrites carries out the rewrite. This module is launched by MonitorRewrites, which determines when the rewrite should execute.
ForceRewrites may be triggered by a user pressing the widget button, Rewrite Outputs. It may also be triggered automatically as follows:
Your custom driver must contain a module named CheckCommLossErr(). An example is provided later in this chapter. This module is called whenever SaveCommStats in VTSDrvr detects an error. CheckCommLossErr() returns a true or false to indicate whether that error qualifies as a loss of communications.
Your custom driver must also contain a Boolean variable named AutoRewrite. If set to TRUE, and if CheckCommLossErr() returns a TRUE value, the trigger is armed.
Upon restoration of communications, as indicated by CheckCommLossErr() returning a FALSE, MonitorRewrites will then execute as follows:
MonitorRewrites monitors the state of the trigger variable, verifies that the server machine's IO instance is in a position to do the writes and verifies that it is the server. Because of the server clause, there will only ever be one instance of this MonitorRewrites through all machines.
The trigger will not be disarmed until ForceRewrite has completed at least one good write.

Read Block Value Coalescing

VTScada combines individual PLC/RTU addresses into a single group or block that is intended to be executed as a single communication message. This is known as "coalescing" and is an attempt to increase throughput by allowing the reading and writing of large blocks of data, where possible.
There are a number of VTScada modules and variables that control how coalescing will be performed:

- VTSGetAddr: Gets the addresses for the blocks of data and parses them. The Info1, Info2, and Info3 parameters provided to this module must be either invalid or the same type in order for coalescing to occur.

- VTSMaxBlock: The value of the VTSMaxBlock variable sets the maximum number of values that can be coalesced into a single read. As an alternative, there may be a VTSMaxBlock subroutine defined, which returns the size of the maximum block for a given address.

The result of coalescing depends on the data type of the MemAddr parameter in VTSGetAddr:

- If MemAddr is numeric, VTScada will coalesce the message blocks into a continuous address range, even if that range includes addresses that are not identified or needed. For example, if the system is configured to read addresses 15 and 50, and the VTSMaxBlock variable is set to 100, then VTSRead is called such that addresses 15 to 50 will be read as a single read message, even though address 16 to 49 inclusive are not needed.
- If the data type of MemAddr is a string, then the result of the coalescing algorithm includes the configured addresses. Using the same example as above, VTSRead is passed an array of strings (in the MemAddr parameter) with two elements 15 and 50.

Communication Driver Design

VTScada was created using an object-oriented, layered approach to software design, and it is highly recommended that VTScada communication drivers follow suit. One advantage to this approach is modularity; components may easily be added or removed without affecting critical components. Layers are designed to build on the functions and services of the lower layers. The communication driver software should be self-contained, and have only a few well-defined entry points. This makes it less reliant on other code so that changes to one part will have a limited effect on other parts of VTScada.

The error state of Communication drivers can be represented on a page by a Status Color Indicator Widget. For the widget to use the error colors defined in an associated Style Settings tag, the driver must contain a flag named ValueIsErrorStatus, set to TRUE. If this flag does not exist, or is

not set TRUE, then the widget will use the colors defined for state 0 and state 1 in the Style Settings tag.
By default, zero is defined as "no error" and all other values as "error". You can expand the range of "no error" values by using the following two parameters:
ValueIsErrorAbove as a numeric. This is the value above which the tag will be treated as being in error. Defaults to 0.
ValueIsErrorBelow as a numeric. This is the value below which the tag will be treated as being in error. Defaults to 0.

Related Information:

...Steps to Write a Communication Driver

...Researching a Communication Driver Protocol

...Designing an Addressing Scheme

...Providing an AddressAssist Module

...Controlling Access to Shared Resources

...Modem Support

...Writing a Communication Driver

...Mandatory Communication Driver Components

...Optional Communication Driver Components

...Data Propagation

Related Functions:

...VTSGetAddr

...VTSRead

...VTSWrite

...VTSMaxBlock

Steps to Write a Communication Driver

1. Gather the details of the hardware protocol.
2. Create a new source file using a text editor (such as UltraEdit). Ensure that this source file is named logically, and is given the extension ".src" (e.g.

"MyDriver.src"). Save this source file on your hard drive where you can easily locate it when it is time to move it to your application directory.

3. Structure your source file according to the rules for a tag template.
4. Referring to the topic, Mandatory Communication Driver Components, add the modules and variables required for a driver. The code you write in the mandatory modules will depend on the driver protocol.
5. According to the details of the driver protocol and your interest in logging statistics, add Optional Communication Driver Components.

Researching a Communication Driver Protocol

A communication driver cannot be written without a full understanding of the protocol that the I/O device uses to communicate. A document that completely describes the protocol must be located, or the device must be reverse-engineered by monitoring communications and experimenting with the inputs and outputs. At a minimum, you will need to know:

- The structure of the messages the I/O device sends and receives.
- The valid values for each component of the message.
- The format that the device expects the data to be in.
- Error detection and correction procedures, if used.
- If a checksum or cyclic redundancy check (CRC) is included in the message, the algorithm that is used to calculate it must be known.

Also, some protocols will define an "end of message" character to indicate the end of the packet. In other protocols, the messages are fixed length, and the length of the packet is either a constant or is indicated as part of the header. You cannot proceed without knowing this structure.

Designing an Addressing Scheme

An addressing scheme is a logical means of organization whereby memory locations in the I/O device are labeled (often, according to the data type stored) for access by external devices. Some protocols specify an addressing structure, and it usually makes sense to use this addressing scheme instead of redesigning it.

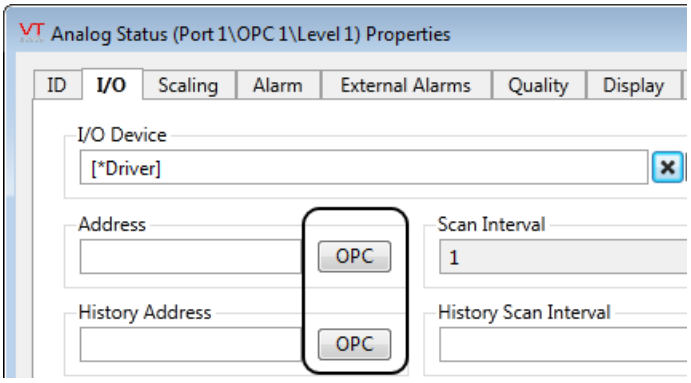
Whatever addressing scheme is used, it must be broken down into information that is stored in a series of 3 parameters (Info1, Info2, and Info3) by a module in the communication driver named, "VTSGetAddr". This information may be the type of data, byte number, record or file number, or something else that is protocol specific. These parameters should contain whatever information is necessary to build and decode the packet that is sent to and received from the correct location in the I/O device.

This information is also used to coalesce the addresses into blocks so that a single communication message can send or receive a large amount of data. Although single address reads and writes are permitted, block reads and writes increase the throughput and overall performance of the software. Addresses are combined into blocks that are as efficient as possible. For example, if the file number is one of the info parameters, the VTSDriver module waits for a series of read or write requests, looks at the file numbers for each request, and then attempts to combine as many sequential file numbers as possible into a single block.

It may be that an address structure is not defined in the device protocol. In such cases, one will have to be designed. An addressing scheme may divide common addresses into groups, but each address must be unique.

Providing an AddressAssist Module

A driver may provide a custom AddressAssist module to assist the user in building addresses when configuring I/O tags. An example may be seen in any I/O tag that is using an SNMP or OPC Client driver.



The module is commonly stored in its own file, with a unique name matching the driver it is meant for, but within the driver itself it must be declared using the name "AddressAssist". The PAddressEntry function (used in the I/O tag's configuration) will check for the existence of an AddressAssist module and (if found) display your code instead of the usual edit field.

The AddressAssist must provide the code for the user interface elements that will be displayed in the I/O tag's configuration panel as well as whatever user-interface tools will be provided to help the developer to select the address. For the configuration panel element, you may:

- draw an edit field with a browse button (as shown in the preceding example),
- create a drop-list or drop-tree with possible addresses,
- create any other address selector that fits within the space of the usual edit-field for address entry.

A simplified example of an AddressAssist module follows.

```
{====================== AddressAssist
==================================}
{ A custom address selection module for a driver. It consists of
}
{ the usual edit field, along with a button that launches an    }
{ address browser.
}
{=================================================================-
=====}
(
  Left                    { The left coordinate of the address
assist    };
  Bottom                  { The bottom coordinate of the address
assist  };
  Right                   { The right coordinate of the address
assist   };
  Top                     { The top coordinate of the address assist
};
  Var                     { The variable that we're going to set
};
  SupportedData           { Digital, Analog, or Text
};
  FunctionType            { Read/Write
};
  ID                      { Focus ID of the editfield
};
  Trigger                 { Set when var changed
};
  BGColor                 { Background colour to use in the edit
field   };
  FGColor                 { Foreground colour to use in the edit
```

```
field   };
)

Main [
  { Display the edit field with a Browse button }
  \System\Edit(Left, Top, Right - SmlBtnWd - Space,
               Bottom + EditHt { Coordinates },
               Invalid, Var, ID, Trigger, Invalid, 4 {Text},
               0 { no bevel }, Invalid, Invalid, Invalid, Invalid,
               Invalid, Invalid, Invalid, BGColor, FGColor);
  If GUIButton(0, 1, 1, 0              { Unit box             },
               1 - (Right - SmlBtnWd)  { 1 - Left             },
               Top + BtnHt             { Bottom               },
               Right                   { Right                },
               1 - (Top)               { 1 - Top              },
               1                       { Scaling              },
               0, 0                    { Movement             },
               1, 0                    { Visibility, Reserved },
               PickValid(ID, 1) != 0 ? 68 : 0, PickValid(ID, 1),
               0  { Selectability },
               ButtonFace,
               ButtonShadow,
               ButtonHighlight,
               ButtonTextColor  { Colors           },
               0, 0             { Sides, Reserved      },
               "...", "..."     { UpLabel, DownLabel   },
               _DialogFont, 0, 1, 2 { Font, DownValue, UpValue, Vari-
able},
               SmallButtonImgs);
  [
    { Implement custom code here to display your address browser }
  ]

  Return(TRUE);
]
```

Controlling Access to Shared Resources

In many cases, an application must communicate with multiple pieces of equipment over a common medium, such as a serial port. In order to prevent multiple driver tags from accessing the same communications medium simultaneously, semaphores (Queued modules) are used. The driver tag is responsible for calling the Sem() module within the VTScada port tag. This module will return true indicating the port is ready to be accessed. It is recommended that driver tags be created with separate read and write semaphore modules. This ensures that queued write requests (such as emergency shut-down instructions) will be processed in a timely fashion by alternating between read and write requests.

Modem Support

Communication drivers are responsible for triggering outgoing modem connections if required. Regardless of whether the driver will be connecting to the device using a modem or not, this mechanism should be implemented as it is required in order to support TCP/IP connections. Also, regardless of whether the hardware supports a TCP connection or not, TCP support should be implemented so that devices such as Lantronix, which provide TCP/IP to serial conversion, can be used.
In order to connect to hardware using a modem, the driver must call the port tag's Port\Connect() module. Before calling the connect module, the driver should check the Port\Modem variable to make sure a connection must be established before communications can begin. The variable Port\IsConnected() can be checked to determine if the connection has been made.
In order for the driver to accept an incoming modem connection ("hardware dialed in"), a discriminator must be written. This module will inform the modem manager which calls should be handed to the driver. This is required since many different units and types of hardware may be dialing in. Details on creating discriminators and registering them with the modem manager are covered in the Modem Manager Service document.

Writing a Communication Driver

Writing a VTScada communication driver involves developing all components that are essential to the software including common driver widgets, error checking, and collection of statistics.
Since VTScada communication drivers are tags, their code must meet all of VTScada's requirements for a tag, as well as the requirements for a driver. The rules for all custom tag types are included elsewhere in the Programmer's Guide. See: Creating Custom Tag Types.
Older drivers often used two module files: one for the high level tasks that interfaced between the I/O tag and the VTSDriver module and one for the lower level tasks that were specific to the driver protocol. This

model is obsolete. It is recommended that communication drivers be created within a single file.

Mandatory Communication Driver Components

Within your driver source file, you will need to create the following modules. These modules are well-defined entry points that are called by VTSDriver in VTSDrvr.web.

- VTSRead – reads VTScada variables from an input device.
- VTSWrite – writes VTScada variables to an output device.
- VTSGetAddr – gets the VTScada addresses and parses them.

You must also define the following variables:

- Driver – This is used internally by the VTScada engine and should not be set by the driver tag code being created. It will automatically be linked to the instance of VTSDriver that will be created for this driver instance.
- Ready – This variable should be set to 1 by the driver tag code when it is safe to start reading or writing I/O.
- Value – This typically defines the error state of the driver where "0" indicates "no error". Value should be set to the Error from VTSRead/VTSWrite, and must be set on the current primary I/O server.
- Root – Set to some object value, usually Self().

Related Information:

...Optional Communication Driver Components

Optional Communication Driver Components

The following variables and modules, while often found in communication drivers, are defined as optional. Of particular note in this list is VTSMaxBlock. While used by nearly every communication driver, it is defined as an optional component.

- Hold – Set to 1 to hold data on error. Otherwise, all data is invalidated when an error occurs.
- VTSMaxBlock – defines the maximum size for block reads and writes

- ByteOrder – reverse byte order if data is not being provided in the Intel, little-endian format. By default, VTScada assumes that the 1st byte is the low-order byte.
- RPCService – necessary for networked applications. Set to a string that is the name of the RPC service that will handle the driver. Usually, the service name will be the same as the tag name. There is also a configuration setting for this value.
- GetClientRevision – RPC service
- GetClientChanges – RPC service
- CommStatsUpdateRate – For this and the next two items, see: Driver Diagnostic Tools.
- CommStatsQualityFactor – see: Driver Diagnostic Tools.
- DisableCommStats – see: Driver Diagnostic Tools.
- StoreOutputs – A variable used to indicate that the driver should store the last set of values written to each output address.
- AutoRewrite – A variable used to indicate that this driver should automatically rewrite stored values upon recovery from a communications error.
- CheckCommsLossErr – A module that determines if the current ErrorVal value indicates a loss of communications. Used if providing AutoRewrite capabilities.

Related Information:

...Mandatory Communication Driver Components

...Driver Diagnostic tools

VTSGetAddr

The purpose of the VTSGetAddr module is to convert the Address parameter (specified in the "Address" field for tags such as analog and digital inputs and outputs) to a value that uniquely classifies the address into a block that can be coalesced by VTScada.
The VTSGetAddr module must be a subroutine. If the given address is invalid, the return value should be an error code.
Format:

VTSGetAddr(Address, MemAddr, BitNum, Info1, Info2, Info3, DataType, Read, Rate);

Parameters:

Address

The raw address from the I/O tag's Address field. This is often a text string, but may be a numeric value. The contents of this parameter are driver-specific. It is the job of this module to interpret the address.
The data type to read from or write to is usually

Suffix	Meaning
/ABFloat	Allen-Bradley PLC/3 Floating Point (4 bytes) (Used for Allen-Bradley exclusively)
/AB5Float	Allen-Bradley PLC/5 Floating Point (4 bytes) (Used for Allen-Bradley exclusively)
/BCD2	2-digit (1 byte) Binary Coded Decimal
/BCD3	3-digit (2 bytes - lowest 12 bits) Binary Coded Decimal
/BCD4	4-digit (2 bytes) Binary Coded Decimal
/Bit	A bit number
/Double	IEEE Double Precision Floating Point (8 bytes)
/Float	IEEE Single Precision Floating Point (4 bytes)
/SByte	Signed Byte
/SDWord	Signed 32-bit Integer
/Sword	Signed 16-bit Integer
/UByte	Unsigned Byte
/UDWord	Unsigned 32-bit Integer

If you intend the address to be numeric, you should ensure this by casting it to a numeric value.

Suffix	Meaning
/ABFloat	Allen-Bradley PLC/3 Floating Point (4 bytes) (Used for Allen-Bradley exclusively)
/AB5Float	Allen-Bradley PLC/5 Floating Point (4 bytes) (Used for Allen-Bradley exclusively)
/BCD2	2-digit (1 byte) Binary Coded Decimal
/BCD3	3-digit (2 bytes – lowest 12 bits) Binary Coded Decimal
/BCD4	4-digit (2 bytes) Binary Coded Decimal
/Bit	A bit number
/Double	IEEE Double Precision Floating Point (8 bytes)
/Float	IEEE Single Precision Floating Point (4 bytes)
/SByte	Signed Byte
/SDWord	Signed 32-bit Integer
/Sword	Signed 16-bit Integer
/UByte	Unsigned Byte
/UDWord	Unsigned 32-bit Integer
/UWord	Unsigned 16-bit Integer

MemAddr

Assists the VTScada code in creating efficient

driver read and write calls. PLC I/O locations with different MemAddr values and the same values for the Info1, Info2, and Info3 parameters are coalesced into the same read block, provided they are within VTSMaxBlock, or the return of VTSMaxBlock(Info1, Info2, Info3, DataType) values of each other.
It is the responsibility of the VTSGetAddr module to set this variable to the correct data type and value.
The MemAddr variable should be one of two data types:

- A pointer to an integer value. If so, it must be set by dereferencing (using a * before the variable name) before being assigned a value.
- A pointer to a string.

VTScada driver code behaves differently, depending on the data type of the variable's contents. This variable is usually a pointer to an integer. If the value is to be a number, make sure a text string is not returned if the value was extracted from the Address string.
If a driver does not use integers to denote an address, return the text string in the MemAddr parameter. The VTSRead module and the VTSWrite module are passed an array of strings rather than a memory starting address. This means, if you set MemAddr to a text value, such as "100" the VTScada code interprets this as a text address rather than a numeric one. This means that the coalescing will not pay any heed to the actual value of the address, but will simply pass an array of text strings to the

VTSRead or VTSWrite in the MemAddr parameter.

BitNumber

A pointer to the integer bit number within the memory address specified by MemAddr. This value may be set to invalid if there is no bit value.

Info1, Info2, Info3

Pointers to values that are used by the VTScada code to determine which addresses can be coalesced into a larger read block. These parameters are also passed to the VTSRead and VTSWrite subroutines. Any values that are not required should be set to "0" (i.e. they must be a valid value). These values are typically set to such things as file type or data area. Any differences between the values of the three Info parameters and those of another Address will prevent those addresses from being coalesced (i.e. grouped into the same read request).

Note: Info1 must be a number; a string value will not work. If your code sets Info1 to a string value, the SetData module in VTSDrvr.web will not work correctly, and client computers will not display the results of the driver server computer's read commands. For example, an analog input that is drawn on the screen will always have invalid contents if the Info parameters are set to strings (such as "abc"), rather than a number.

DataType

A pointer to a list of values that holds the data types preferred for the specified address. This data type may be left invalid if the VTSRead and VTSWrite subroutines

ignore it. If left invalid the default data type will be 2 (16-bit unsigned integer). This value will be overridden by the explicit data type specified as an appended string to the address. The DataType is what is passed to the VTSRead and VTSWrite subroutines. The appended data type (refer to the table in the Address parameter section above) specifies how the data is to be interpreted after it is read from the I/O device if this is to be different from the DataType returned by VTSGetAddr. The VTSRead and VTSWrite modules do not see the value of the appended data type string.

Note: The MemAddr, BitNum, Info1, Info2, Info3, and DataType parameters are pointer values, and must be set by dereferencing them before assigning the values (i.e. using an asterisk "*" before the variable name (please refer to "Pointers" for further details on pointers and dereferencing).

Read

Specifies whether the address will be used for a Read or a Write command. If true, the address is used for reading data, otherwise it is used for writing data. Most drivers ignore this parameter since there is usually no difference in the returned values for reads and writes.

Rate

A pointer to the scan rate of the I/O device. This may be an Invalid pointer.

Comments:

The VTSGetAddr subroutine must return an error code. If the return value is not 0, the VTScada code assumes that there is an error in the address and it will not include that tag when coalescing the block.

VTSRead

The VTSRead module is run continuously to read data from the I/O device. It is called from the VTSDriver code.
A VTSRead should return data for one or more addresses as contained in the array parameter, MemAddress. Data is returned by calling CallBack-Obj\RefreshData as described in the topic, Data Propagation. An example is provided at the end of the parameter description.
Format:
VTSRead(Trigger, Data, N, MemAddr, BitNumber, Info1, Info2, Info3, DataType[, CallBackObj]);
Parameters:

Trigger

A logical value that, when true, requests that the VTSRead module scan the I/O device for data.

Data

An array into which the read data is to be placed.

N

The number of values to read from the I/O device. If MemAddr is an array of strings, then N is set to the size of the array.

MemAddr

Assists VTScada in creating more efficient driver read and write calls. PLC I/O locations with different MemAddr values and the same values for the Info1, Info2, and Info3 parameters are coalesced into the same read block, provided they are within VTSMaxB-lock, or the return of VTSMaxBlock(Info1, Info2, Info3, DataType) values of each other.
It is the responsibility of the VTSGetAddr module to set this variable to the correct data type and value. Note that the data type of the MemAddr parameter in VTSGetAddr affects the data type of this parameter. If

the data type is a string in VTSGetAddr, then this parameter will be an array of strings.

BitNumber

A pointer to the integer bit number within the memory address specified by MemAddr. This value may be set to invalid if there is no bit value.

Info1, Info2, Info3

Are variables (not pointers), that are used by the VTScada code to determine which addresses can be coalesced into a larger read block. These parameters are also passed to the VTSGetAddr and VTSWrite subroutines. Any values that are not required should be set to "0" (i.e. they must be a valid value). These values are typically set to such things as file type or data area. Any differences between the values of the three Info parameters and those of another Address will prevent those addresses from being coalesced (i.e. grouped into the same read request).

Note that Info1 must be a number: a string value will not work. If your code sets Info1 to a string value then the SetData module in VTSDrvr.WEB will not work correctly and client computers will not display the results of the driver server computer's read commands. For example, an analog input that is drawn on the screen will always have invalid contents if the Info parameters are set to strings rather than a number.

DataType

A pointer to a value that holds the data type preferred for the specified address. This data type may be left invalid if the VTSRead and VTSWrite subroutines ignore it. If left invalid the default data type will be 2 (16-bit unsigned integer).
This value will be overridden by the explicit data type

specified as an appended string to the address. The DataType is what is passed to the VTSRead and VTSWrite subroutines from VTSGetAddr. The appended data type (refer to the table in the "Address" parameter section above) specifies how the data is to be interpreted after it is read from the I/O device if this is to be different from the DataType returned by VTSGetAddr. The VTSRead and VTSWrite modules do not see the value of the appended data type string.

CallBackObj

Optional, but recommended. An object value whose scope is used to call RefreshData. (see: Data Propagation) RefreshData is the recommended method for propagating data.

If using RefreshData to propagate data, then VTSRead should not also update the Data Array parameter on its own.

VTSRead must return an object (usually Self()) that includes the three mandatory variables: Counts, ErrorCounts and Error.

Note: VTSRead must increment either Counts or Error Counts before processing new data.

A typical example of the code for the VTSRead subroutine:

```
<
VTSRead
(
  Trigger{ Will do read on positive edge          };
  Data{ Array to put the result into              };
  N{ The number of values to read                 };
  MemAddress{ Address to read                     };
  BitNum{ Bit to read / not used here             };
  Info1{ Info1                                    };
  Info2{ Info2                                    };
  Info3{ Info3                                    };
  DType{ Data type used for read                  };
  CallBackObj{ Object value where RefreshData is located };
)
[
  Error        = 0;
  Counts       = 0;
  ErrorCounts  = 0;
```

```
  StartTime               { Request start time              };
  ElapsedTime             { Request elapsed time            };
  Tries        = 0        { Number of attempts on modem     };
]
Main [

  If Trigger;
  [
    ResetParm(Self(), 1) { Reset the trigger since does not feed back
                           from the driver };

      { Process data according to the protocol }
    . . .

      { error in data, increment ErrorCounts },
    IfElse((AValid(TimeArray[0], ArraySize(TimeArray, 0) != ArraySize
(TimeArray, 0)) ||
           (AValid(DataArray[0], ArraySize(ValueArray, 0) != ArrayS-
ize(ValueArray, 0)),
      ++ErrorCounts,
    {Else - no errors in data, increment Counts and send data}
      Execute(
        ++Counts,
          { Send data and timestamps }
        CallBackObj\RefreshData(TimeArray,ValueArray,Invalid,Invalid)
      ) {End Execute},
    ) {End IfElse};

    Return(Self);
  ]
]
>
```

Data Propagation

The VTSRead module is responsible for delivering the data it read from the I/O device. This is best done by using the RefreshData module within the scope of the CallBackObj parameter of the VTSRead module.
The data type and structure can vary for the three parameters, NewData, TimeStamp and Attribute. Read the following notes carefully. If the driver will read blocks of history data, please refer to the discussion of Block History in the Comments section of this topic.
RefreshData is provided for you as part of the VTSDriver module.
Format:CallBackObj\RefreshData(TimeStamp, NewData, Attribute, QueueObj, PropagateOnlyOnDataChange)
Parameters:
TimeStamp Optional. If invalid, the system uses current time in UTC to timestamp the data. Otherwise, may be either of the following:

- A single timestamp value, expressed in number of seconds since January 1st 1970 UTC. This value will be applied to each element of NewDta.
- An array, having the same structure as NewData. Each data point will be assigned its corresponding timestamp from the timestamp array.

For event driven reads, you may wish to process only the data from a single address, rather than from a full read block. You can achieve this by setting all the TimeStamp array elements to invalid except for the one matching the element of NewData which you want to process.
Note that invalid timestamps in a timestamp array have special meaning. An invalid timestamp element indicates that the corresponding data has not changed. This enables drivers to pass back partial updates. This feature is independent of the PropagateOnlyOnDataChange option.
NewData This is the value read from the I/O device. It may be either a simple array, or an array of arrays having the same size as the MemAddress array used in the VTSRead function.
If a simple array, there will be one element for each address. The array of arrays option is generally used for returning historical data for a given address.
Calling Refresh data with the NewData parameter specified as an array of arrays will work only with Analog Status, Digital Status and Pump Status. It will not work with Analog Input, Digital Input, and any tag not specifically coded to handle an array of results being passed into newData().
Attribute Optional. No default. May be a single value, which will be applied to all elements of NewData if that is an array, or may be an array of attributes that gets assigned to the corresponding array of data.
If an array, the size and arrangement of the array must be the same as NewData.
QueueObj Obsolete. Should be set to Invalid.
PropagateOnlyOnDataChange Flag that when set true, will prevent data from being propagated through RPC or to tag\NewData() unless the value of NewData or Attributes has changed since the last time RefreshData was called. Defaults to TRUE.

Historical reads, whether by array or array-of-array, are expected to set PropagateOnlyOnDataChange to FALSE in order to disable the automatic filtering. One can interpret this option as 'automatically detect and ignore data which has not changed'.

Comments:

Invalid data is accompanied by a valid timestamp:

```
TimeStamp[i] = Some Valid Time;
Data[i]              = Invalid;
```

Missing Data is indicated by an invalid timestamp:

```
TimeStamp[i]  =  Invalid;
```

Block history

When reading historical data, an array of arrays may optionally be used, where each element of the NewData array will contain another array which must be sized for the number of data points that are being returned for the matching MemAddress. This holds true even if there is only one value to return for each I/O address.

This a special mode used only for input tags that can support the different read result. These are, the AnalogStatus, DigitalStatus and the PumpStatus, all of which have a 'History Address' parameter. The reading of block history has little to do with VTSDriver itself since VTSDriver simply passes through whatever the driver provides. The contract that standard status tags expect is that a block of history will be passed to NewData() as an array, and that the Timestamp parameter will be a matching array with corresponding timestamps. If Attributes are present, then attributes for block history should be in an array corresponding to the Data/Timestamp arrays. It is important that drivers pass block history data in the form that VTScada status tags support.

If the block of history records is being passed in an array, then the oldest timestamp must be at index 0. The restriction that the block history be in a certain order is due to the Historian, which logs exactly what it is given without modification (by design). Having timestamps reversed means that out-of-order history is written, which will slow performance. If the

driver's protocol requires block history with the newest item first, then that driver should swap the order before passing it along.

Example 1:

Given MemAddress with two addresses, "Addr1" and "Addr2" and data for Addr1 = 1.23 and data for Addr2 = 2.34, you might do something like:

```
{ Make a data array }
Data = New(2);
 { Add first element }
Data[0] = New(1);
Data[0][0] = 1.23;
 { Add second element }
Data[1] = New(1);
Data[1][0] = 2.234;

{ Make it So }
CallBackObj\RefreshData( Invalid, Data, Invalid );
```

Example 2:

Multiple Data/Timestamp Value Pairs for each item in MemAddress, no Attributes

Given MemAddress with two addresses, "Addr1" and "Addr2" and data/-timestamps for

Addr1 = 1.23/2009-08-01 11:23, 1.24/2009-08-01 11:24 and data for Addr2 = 2.34/2009-08-01 11:21, 2.35/2009-08-01 11:22

you might do something like:

```
{ Make a data array }
Data      = New(2);
TimeStamp = New(2);

{ Add first element}
TimeStamp[0]   = New(2);
Data[0]   = New(2);
TimeStamp[0][0] = ConvertTimestamp(
                     \ODBCManager\ConvertToVTSTimeStamp(
                        "2009-08-01 11:23"),
                        "US Eastern Standard Time", 0,
                        "GMT Standard Time" );
Data[0][1]      = 1.24;
TimeStamp[0][1] = ConvertTimestamp(
                     \ODBCManager\ConvertToVTSTimeStamp(
                        "2009-08-01 11:24"),
                        "US Eastern Standard Time," 0,
                        "GMT Standard Time" );

{ Add second element}
```

```
Data[1]      = New(2);
TimeStamp[1] = New(2);

Data[1][0]      = 2.34;
TimeStamp[1][0] = ConvertTimestamp(
                     \ODBCManager\ConvertToVTSTimeStamp(
                        "2009-08-01 11:21"),
                        "US Eastern Standard Time," 0,
                        "GMT Standard Time" );
Data[1][1]      = 2.35;
TimeStamp[1][1] = ConvertTimestamp(
                     \ODBCManager\ConvertToVTSTimeStamp(
                        "2009-08-01 11:22"),
                        "US Eastern Standard Time," 0,
                        "GMT Standard Time" );

{ Make it So }
CallBackObj\RefreshData( TimeStamp, Data, Invalid );
```

VTSWrite

The VTSWrite module is launched by the VTSDriver code to write a block of values to the I/O device. Its code is very similar to that of the VTSRead module with the same number of parameters.
The VTSWrite module will usually be slain immediately after the write is completed.
Format:
VTSWrite(Trigger, Data, N, MemAddr, BitNumber, Info1, Info2, Info3, DataType);

Parameters:

Trigger

A logical value that, when true, will request that the VTSWrite module send the data to the I/O device.

Data

An array where the written data will be read.

N

The number of values to write to the I/O device. If MemAddr is an array of strings, then N indicates the size of the array.

MemAddr

Assists VTScada in creating more efficient driver read and write calls. PLC I/O locations with different MemAddr values and the same values for the Info1, Info2, and Info3 parameters are coalesced into the same write block, provided that they are within VTSMaxBlock, or the return of VTSMaxBlock(Info1, Info2, Info3, DataType) values of each other.
It is the responsibility of the VTSGetAddr module to set this variable to the correct data type and value. The data type of the MemAddr parameter in VTSGetAddr affects the data type of this MemAddr parameter. If the DataType is a string in VTSGetAddr, then this parameter will be an array of strings.

BitNumber

A pointer to the integer bit number within the memory address specified by MemAddr. This value may be set to invalid if there is no bit value.

Info1, Info2, Info3

Are variables (not pointers) that are used by the VTScada code to determine which addresses can be coalesced into a larger read block. These parameters are also passed to the VTSRead and VTSWrite subroutines. Any values that are not required should be set to "0" (i.e. they must be a valid value). These values are typically set to such things as file type or data area. Any differences between the values of the three Info parameters and those of another Address will prevent those addresses from being coalesced (i.e. grouped into the same read request).

Note that Info1 must be a number: a string value will not work. If your code sets Info1 to a string value, the SetData module in VTSDrvr.WEB will not work correctly, and client computers will not display the results

of the driver server computer's read commands. For example, an analog input that is drawn on the screen will always have invalid contents if the Info parameters are set to strings rather than a number.

DataType

A pointer to a value that holds the data type preferred for the specified address. This data type may be left invalid if the VTSRead and VTSWrite subroutines ignore it. If left invalid the default data type will be 2 (16-bit unsigned integer). This value will be overridden by the explicit data type specified as an appended string to the address. The DataType is what is passed to the VTSRead and VTSWrite subroutines from VTSGetAddr. The appended data type (refer to the table in the Address parameter section above) specifies how the data is to be interpreted after it is read from the I/O device if this is to be different from the DataType returned by VTSGetAddr. The VTSRead and VTSWrite modules do not see the value of the appended data type string.

VTSWrite must increment one of two mandatory internal variables each time it is called. Either "Counts" if the write was successful or "ErrorCounts" otherwise.

The code for VTSWrite might look like:

```
<
VTSWrite
(
  TriggerParm{ Will do read on positive edge                 };
  Array{ Array to put the result into                    };
  N{ The number of values to read                  };
  MemAddress{ Address to read                              };
  BitNum{ Bit to read                                 };
  Info1{ Info1 - not used                              };
  Info2{ Info2 - not used                              };
  Info3{ Info3 - not used                              };
  DataType{ Data type used for read - uses standard types };
)
[
    { Trigger is PUBLIC and is referenced by VTSDriver }
  Trigger= 1{ TRUE when initial trigger set                };
  Error        = 0;
```

```
  Counts       = 0;
  ErrorCounts  = 0;
  StartTime          { Request start time                       };
  ElapsedTime        { Request elapsed time                     };
  Tries        = 0   { Number of attempts on modem              };
  WriteActive  = 0   { TRUE when the write has been triggered   };
]

Main [

  If Trigger || TriggerParm;
  [
      { Process the data according to the protocol }
   . . .

{ Reset the triggers }
    ResetParm(Self(), 1);
    Trigger = 0;
  ]
]
>
```

VTSMaxBlock

VTSMaxBlock determines the maximum block size that is coalesced into a VTScada read or write. This can be either a subroutine or a variable.

- If the maximum block size is a constant, regardless of the data type or address, then VTSMaxBlock can be declared as a variable and set to a numeric value.
- If supplied as a subroutine then, based on the address and data type information, this module should return the maximum amount of data that can be handled in a single read or write. It is your job to write this subroutine as part of your driver, using the following format:

Format:

VTSMaxBlock(Info1, Info2, Info3, DataType);

Parameters:

Info1, Info2, Info3

are pointers to values that are used by the VTScada code to determine which addresses can be coalesced into a larger read block. These parameters are also passed to the VTSRead and VTSWrite subroutines. Valid values must be supplied for all three, therefore any values that are

not required should be set to "0".

These values are typically set to such things as file type or data area. Any differences between the values of the three Info parameters and those of another Address will prevent those addresses from being coalesced into the same read request.

Note: The value of Info1 must be numeric. If your code sets Info1 to a string value, the SetData module in VTSDrvr.WEB will not work correctly and client computers will not display the results of the read commands.

DataType

A pointer to a value that holds the data type preferred for the specified address. This data type may be left invalid if the VTSRead and VTSWrite subroutines ignore it. If left invalid the default data type will be 2 (16-bit unsigned integer). This value will be overridden by the explicit data type specified as an appended string to the address. The DataType is what is passed to the VTSRead and VTSWrite subroutines from VTSGetAddr. The appended data type (refer to the table in the Address parameter section above) specifies how the data is to be interpreted after it is read from the I/O device if this is to be different from the DataType returned by VTSGetAddr. The VTSRead and VTSWrite modules do not see the value of the appended data type string.

Communication Driver Template

A sample communication driver is displayed in part here. Code that is useful only for the particular hardware this driver was designed to communicate with is not included. This particular driver was designed to be read-only. It does not contain a VTSWrite module. The tag's configuration and common modules are not shown here.
The driver starts with standard tag definitions:

```
{=============================================================}
(
  Name              <:TagField("SQL_VARCHAR(64)",  "Name" ):>;
  Area              <:TagField("SQL_VARCHAR(255)", "Area" ):>;
  Description       <:TagField("SQL_VARCHAR(255)", "Description" ):>;
  RespTimeOut       <:TagField("SQL_VARCHAR(255)", "RespTimeout" ):>;
  SiteID            <:TagField("SQL_VARCHAR(255)", "SiteID" ):>;
  UTCOffset         <:TagField("SQL_VARCHAR(255)", "UTCOffset" ):>;
  HelpKey           <:TagField("SQL_VARCHAR(255)", "HelpKey" ):>;
…  StoreOutputs      <:TagField("SQL_VARCHAR(255)" ):>
          { When TRUE permits storing of output values     };
  AutoRewrite       <:TagField("SQL_VARCHAR(255)" ):>
          { When TRUE permits the automatic Rewriting of Outputs };
)
[ {Variables}
  {=================Version Control Information ================}
  { V 0.0.01 - Original issue - 9 December, 2008                 }
  {=============================================================}
  Constant DriverVersion                  = "0.10.0 6 January, 2009";
  Constant DrawLabel                      = "USGSDriver";
  Constant #Name                          = 0;
  Constant #Area                          = 1;
  Constant #Description                   = 2;
  Constant #ResponseTimeOut               = 3;
…
```

In the variable declaration modules and error constants are also defined:

```
  { Module Numbers }
  Constant #VTSGetAddr                    = 0;
  Constant #VTSRead                       = 1;

  { Error Constants }
  Constant #NoError                       = 0;
  Constant #ConnectionError               = 1;
…
```

As well as the required modules and variables (plus a few extra shown here):

```
  {***** Required for all VTScada drivers *****}
  DriverNameLabel     = "DemoDriver";
```

```
  Driver                { The generic driver module instance   };
  Ready                 { Indicates to VTScada driver the module is
ready };
  RPCService            { Name of the RPC service for this tag };
  Value                 { Driver status                        };

{ Modules }
  ShowComm      Module  "DemoShowComm.SRC"  { Show communication };
  ShowStats     Module  "DemoShowStats.SRC" { Show comm statistics };
  VTSMaxBlock   Module  { Returns Maximum record size          };
  VTSRead       Module  { The Read module                      };
  VTSGetAddr    Module  { Subroutine to parse address          };
  Refresh       Module  { Refresh module                       };
  ErrMessage    Module  { Converts error codes to test strings };
  ReportTraffic  Module { Translator for the traffic monitor   };
  SetStats      Module  { Updates driver statistics            };
  TransmitReceiveData Module { Transmit/Receive Module         };
  ProcessReturnedData Module { Processes ret data into arrays  };
  SiteTime_2_UTCTS    Module { Converts Site time to VTScada time  };
  HistPoll     Module    { Sets the history poll values        };
  ReadLineStatus  Module  { Reads a line and returns status    };
  CheckCommsLossErr Module { Checks the comms fail errors      };

   { Variables }
  Root              { Root object value                    };
  DriverObj         { Object value of driver               };
  CommPortObj       { Communication port tag object        };
  ErrorMessages     { Array of Error Messages              };
  Counts       = 0  { # of successful transactions         };
  TimeStamp         { Time of last successful transaction  };
  Error          = 0 { Global Error code                    };
  ErrorCounts    = 0 { # of errors since starting           };
  ConsecErrCount = 0 { Consecutive time out error counts    };
  ErrorMemAddr       { Memory address of last error         };
  ErrorTime          { Time of last error                   };
  LastError          { Error code for last error            };
  Shared Message[100]  { List of error messages             };
  CommDisplay        { Object communications display module };
  ErrorModule        { Module number generating error       };
  ErrorDate          { Date of last unsuccessful com attempt};
  ErrorOwner         { Object value of module last error    };
  dt                 { Time since last transaction          };
  DateStamp          { Date of last successful comm.     };
  LastURL            { Last URL generated                   };
  ResponseTO         { Response Time Out variable           };
  DataFilePath       { Path for data file to write/read     };
  Status             { Status of file directory creation    };
```

Finally, the standard groups memberships are defined:

```
  { Parameter Constants }
  [ (GROUPS)
   Shared Numeric;
   Shared Drivers           { This is a driver point };
  ]

  [ (GRAPHICS)
   Shared CommIndicator;
```

```
   Shared CommStatistics;
   Shared CommMessages;
   Shared ReWriteOutputsBtn; { Used if implementing Rewrite Outputs }
  ]

  [ (PLUGINS)
    Shared ConfigFolder = "DemoDriverConfig";
    Shared Common       = "DemoDriverCommon";
  ]
```

The module opens with an initialization state:

```
DemoDriverInit [

  If \NetworkValues\Started WaitServer;
  [
     { Set the RPC service for this instance depending on the }
     { Configuration parameter }
    RPCService = PickValid(\USGSSharedRPC, 0) ? "USGSDriver" : Name;
     { Return object value }
    Root = Self();
    DriverObj = Root;
    Refresh();

    IfThen(!Valid(Message[#NoError]);
       { Set up of driver statistics display labels }
      CountsLabel           = PickValid(\CountsLabel, "Counts");
       { … etc … }

       { Driver internal errors }
      Message[#NoError                ]  = "No Error";
       { … etc … }

    ) { IfThen };

     { Register with the network values service }
    \NetworkValues\Register(Self(), Name);

  ]
]
```

Rather than one main state, the driver has three: Wait, Client and Server. Most of the time it will be in the Wait state, waiting to send or receive data.

```
Wait [
    { If this PC is a data server for this PLC, }
    { go to the Server state }
  If PickValid(Driver\Started, 1) &&
     PickValid(*(Driver\RPCStatus), \#RPCServer { Server }) ==
\#RPCServer Server;
    { If this PC is a client for this PLC, go to the Client state }
  If PickValid(Driver\Started, 1) &&
             *(Driver\RPCStatus) == \#RPCClient Client;
]
```

```
Server [
    { If no longer server go to wait state }
  If *(Driver\RPCStatus) != \#RPCServer Wait;
    { Get object values in steady state }
  CommPortObj = USGSTCPIPPort;
  Ready = 1;
    { Reset if there is a switch from a server to a client }
  If *(Driver\RPCStatus) != \#RPCServer Wait;
]

Client [
    { Get object values in steady state }
  Ready = 1;

  If PickValid(*(Driver\RPCStatus),\#RPCServer) != \#RPCClient Wait;
]
```

The last tag-specific part of the driver is the Refresh module. If implementing Auto-Rewrites, include code to ensure that the StoreOutputs value is always true if AutoRewrite is true.

```
<
{========================== Refresh ============================}
{ Refresh subroutine.                                           }
{===================================================================}
Refresh
(
  Parm { Array parameters prior to their change }
)

Refresh [
  If Watch(1);
  [
    RPCService = Name       { Set RPCService                  };
    UTCOffset  = PickValid(Cast(UTCOffset, 2) , TimeZone(0));
    ResponseTO = PickValid(ResponseTimeOut, 10);

   StoreOutputs = PickValid(Cast(StoreOutputs, \#VTypeShort), 0);
   AutoRewrite  = PickValid(Cast(AutoRewrite,  \#VTypeShort), 0);
{ Sort the parms, when AutoRewrite is true, StoreOutputs must be true
}
   IfThen(AutoRewrite && !StoreOutputs,
     StoreOutputs = 1;
   );


   IfThen(!StoreOutputs &&
           PickValid(*(Driver\RPCStatus) == \#RPCServer, 0),
      Driver\ResetOutputDict();
   );

    Return(0);
  ]
]
{ Refresh }
>
```

Refresh is followed by the standard modules of a communication driver: VTSGetAddr, VTSMaxBlock, VTSRead and VTSWrite. In the following examples, any code that is not general in purpose has been removed.

```
<
{================ USGSDriver\VTSGetAddr ======================}
{                                                              }
{==============================================================}
VTSGetAddr
(
  Address        { Raw address specified by point parameters};
  RetAddress     { Returned address                         };
  BitNum         { Pointer to bit number variable to set    };
  TableName      { Table name                               };
  Info2          {                                          };
  Info3          {                                          };
  DType          { Pointer to data type to return           };
  Read           { TRUE if a READ address, otherwise write  };
  Rate           { Rate for read coalescing                 };
)
[
  NRead          { Number of Values not read                };
  SiteNumber     { Station Number Field                     };
  ParameterField { Parameter Field                          };
  AddressError   { Set on error                             };
]

Main [

  If Watch(1);
  [
    AddressError = 0;
     { Need an valid non-null value address }
    IfElse(Valid(Address) && StrLen(Address) > 0;
       { Set to incoming address -          }
       { sends array of addresses to VTSRead }
      *RetAddress   = Address;
      *AddressError = 1;
    ); { IfElse }
     { Return error }
    Return(AddressError);
  ]
]
{ End of VTSGetAddr }
>

<
{====================== VTSMaxBlock ========================}
{ Returns the Maximum size for block reads and writes. In    }
{  example, VTSMaxBlock will simply return a constant.    }
{==============================================================}
VTSMaxBlock
(
  Info1              { Not used                 };
  Info2              { Not used                 };
  Info3              { Not used                 };
```

```
  DType              { Pointer to data type      };
)
[
 Constant #VTSMaxBlock = 100;
]

VTSMaxBlock [
  Return(#VTSMaxBlock);
]
{ VTSMaxBlock }
>
```

Here comes the VTSRead module. Again – only the most general purpose of statements have been included.

```
<
{================= USGSDriver\VTSRead ======================}
{ This module performs the I/O reads.                    }
{=========================================================}
VTSRead
(
  Trigger       { Will do read on positive edge             };
  Array         { Array to put the result into              };
  N             { The number of values to read              };
  MemAddr       { Field Name Parameter                      };
  BitNum        { Pointer to bit number variable  (not used)};
  Info1         { Site Number to read                       };
  Info2         {                                           };
  Info3         {                                           };
  DType          { Data type                                 };
  RefreshContext { Where to call the RefreshData() module    };
)

[
  Name   = "VTSRead"  { Module name for stats                };
  LocErr              { Local Error                          };
  Counts       = 0    { # of successful transactions         };
  TimeStamp           { Time of last successful transaction  };
  Error        = 0    { Global Error code                    };
  ErrorCounts  = 0    { # of errors since starting           };
  ErrorMemAddr        { Memory address of last error         };
  ErrorTime           { Time of last error                   };
  LastError           { Error code for last error            };
  ModNumber           { Module number for VTScada read           };
  SiteData            { Data array to pass to refresh context};
  SiteTimeStamp       { Timestamp array to pass to refresh context};
…
]

Init [
  If 1 VTSWaitTrigger;
  [
    Return(Self);
  ]
]

VTSWaitTrigger [
```

```
  If Trigger GetSiteData;
  [
    ResetParm(Self(), 1)      { Reset the trigger              };
    LocErr         = Invalid();
    SiteData       = New(N);
    SiteTimeStamp  = New(N);

     { Type of poll - default to data since last poll }
    PollType       = PickValid(HistPollType, #HistPollSinceLast);
  ]
]

GetSiteData [
  LocErr = TransmitReceiveData(SiteID, MemAddr, StartDateTS,
                               EndDateTS);
  If !LocErr ProcessData;
  [
    LocErr = Invalid();
  ]
  If LocErr SetStats;
]

ProcessData [
  LocErr = ProcessReturnedData(&SiteData, &SiteTimeStamp,
                               LastTSRead);
  If !LocErr SetStats;
  [
     { Send the new data and timestamps }
    RefreshContext\RefreshData(SiteTimeStamp, SiteData);
     { Update the network variable }
    LastReadingTS = Max(PickValid(LastReadingTS, 0), LastTSRead);
  ]
  If LocErr SetStats;
]

SetStats [
  If 1 VTSWaitTrigger;
  [
    SetStats(Error = LocErr, Self(), ModNumber, 0, 0)
                    { Set driver stats };
    IfElse(LocErr;
      ++ErrorCounts;
       { Else no error }
      ++Counts { Increment counts };
    ) { IfElse };
  ]
]
{ USGSDriver\VTSRead }
>
```

If implementing automatic rewrites of saved data, you must have Check-CommsLossErr() module similar to the following. The specific error codes will depend on your driver – those used in the example are for the Allen Bradley driver only.

```
<
{================= ABDriver\CheckCommsLossErr ===================}
{================================================================}
CheckCommsLossErr
 (
  ErrorVal;
)
Main [
  If 1;
  [
    {***** Comms Loss errors noted for the AB driver *****}
    IfThen(ErrorVal == 4       ||
           ErrorVal == 32      || ErrorVal == 48    ||
           ErrorVal == 64      || ErrorVal == 0x200 ||
           ErrorVal == 0x202   || ErrorVal == 0x213 ||
           (ErrorVal >= 0x300  && ErrorVal <= 0x309),
      Return(1);
    );
    Return(0);
  ]
]
{ End of ABDriver\CheckCommsLossErr }
>
```

In the code for your driver's configuration panel, if you are implementing output rewrites, you must also provide a check box for AutoRewrite and StoreOutputs. Include the following code to ensure that if AutoRewrite is true, StoreOutputs can't be false:

```
  If Watch(0, Parms[\#StoreOutputs]);
  [
    IfThen(!Parms[\#StoreOutputs],
  { Ensures if StoreOutputs is false then AutoRewrite can't be true }
      Parms[\#AutoRewrite] = 0;
    );
  ]

  If Watch(0, Parms[\#AutoRewrite]);
  [
    IfThen(Parms[\#AutoRewrite],
  { Ensures if AutoRewrite is true then StoreOutputs can't be false }
      Parms[\#StoreOutputs] = 1;
    );
 ]
```

The VTSDriver API

The VTSDriver module is supplied with VTScada. It includes the generic driver interface for VTScada and is used in combination with each

device-specific communication driver.

Several of the functions in VTSDriver are called directly from I/O tags. For example, AddRead is called by an input or status tag to request that data be sent from the driver. AddRead will call VTSRead in the device-specific communication driver to do the actual work according to the particular device's requirements.
Note that you cannot call VTSDriver\PollAll() directly after calling AddReads(). AddReads takes a non-zero time to start, but PollAll synchronously triggers all current reads. The solution is to wait for VTSDriver\RefreshReady to become TRUE after calling AddRead. VTSDriver\RefreshReady is FALSE when read modules are being changed (in flux), and TRUE once they have stabilized on their new state. AddRead and DelRead set it FALSE, and it is then set TRUE again once the Read modules are up and running.

Started A flag to let you know that RPC is ready.

RPCStatus Provides the client/server state.

AddRead A subroutine called by I/O tags to request a read from the communication driver. Does not force a read to occur immediately. Parameters:

Address

The address from which to get the data.

N

The number of elements to get

Value

Either a pointer to the destination for the data or an object value if VTSDriver will call Value\NewData when data changes.

Rate

The data update period, measured in seconds.

NewData Typically called from within an I/O tag when RefreshData is called and before the data is returned. NewData will only be called when there is new data available – including invalids.

Parameters:

Address

From the original AddRead

Time

StampIn seconds, UTC

Data

The new data – either a single value or an array of values.

Attribute

Data attributes. Single value or array, matching Data.

DelRead A module that must be called when the address changes.

Read A module that performs a one-shot read, the data from which will be sent only to the calling server.

Write The main subroutine used to request a write to the communication driver.

AddWrite *Obsolete*. Was once used to request a write to the communication driver.

PollAll Forces all pending reads to occur immediately. This subroutine, which includes feedback, is especially useful with radio links where reads must occur when the link is established.

CoalesceRPC Obsolete. Do not use in new code. Remove from code being upgraded to the current version of VTScada.

Related Information:

...VTSDriver and Remote Applications

Related Functions:

...VTSGetAddr

...VTSRead

...VTSWrite

...VTSMaxBlock

VTSDriver and Remote Applications

If you are running a remote application, you should be aware of the following details:

- The subroutines VTSRead and Write will not run on client workstations.
- I/O addresses that are configured only on a client workstation will not be read. They must be committed to the server before they will be used.
- Data blocks are synchronized on startup.
- Switching can occur on RecommendAlternate in order to provide for soft fail-overs.
- Drivers sharing the same serial port must be in the same RPC service. The RPCService variable sets the service name for a driver instance.

Driver Diagnostic tools

The VTScada driver module contains a traffic monitor API, ReportTraffic, allowing the driver communications stream to be viewed or logged by viewer applications. The interface is accessed through the TMObj variable within the scope of the "Driver" Object value. Traffic must be manually reported by the driver tag using:
Format: Driver\TMObj\ReportTraffic(PortName, Direction, TrafficData);
Parameters:

PortName

This should be set to a text string identifying the port tag used by the driver.

Direction

Specifies the direction of the traffic (0 = Receive, 1 = Transmit).

Traffic

Data Text string used to display the actual contents of the communications data. This string will be displayed as is and should be formatted into a legible string before being passed to the ReportTraffic Module.

The "Active" flag located within the scope of TMObj indicates whether any listener applications are monitoring the driver. ReportTraffic should only be executed if the "Active" flag is set.

```
IfThen(Driver\TMObj\Active,
  Driver\TMObj\ReportTraffic(ExternalPort\Name, Direction, Data);
);
```

Related Information:

...Statistics Logging

...Error Checking

...Debugging and Testing Communications Drivers

Statistics Logging

The VTScada driver module contains an API that is used to log statistics regarding the status of the communications driver. The SaveCommStats subroutine is located in the scope of the "Driver" variable. The communications driver is responsible for calling this subroutine whenever a successful or a failed communications attempt has been detected. In other words, anytime the error value of the driver is set.
Format:
Driver\SaveCommStats(ErrorValueParm, ErrorAddressParm, ErrorAfterRetryParm, ResponseTimeParm, CommStatsCallerInfoParm);
Parameters:

ErrorValueParm

This parameter should be set to the current numerical error value of the communications driver. A value of 0

indicates success.

ErrorAddressParm

This should be set to the memory address which caused the error to occur.

ErrorAfterRetryParm

This is a flag indicating whether or not the error, or the success, occurred on a retry. A value of 1 indicates this was a retry attempt, a value of 0 indicates this was the initial attempt.

ResponseTimeParm

Expressed in seconds, this parameter specifies the amount of time it took the I/O device to receive a command, process it, and send a response. This should only be valid if the ErrorValueParm is 0, meaning a successful communication.

CommStatsCallerInfoParm

An optional parameters allowing the I/O driver to pass a string which can be displayed in a real time Statistics viewer. This value is not logged, and will fault to the module name of the SaveCommStats caller.

The ResponseTime is defined as the time it took the I/O device to receiver a command, process it, and return a response. The I/O driver is responsible for calculating the response time. It should be calculated using the following formula:

```
ResponseTime = EndTime - StartTime - XmitTime - RcvTime
```

Where XmitTime is the time it took to transmit any data and RcvTime is the time it took to receive the data based on the number of bytes sent, the baud rate, parity, stop bits, whether there is an echo expected and whether RTS key delays are used. It is the responsibility of the communications driver to calculate the ResponseTime.

Communications driver (driver tag) statistics variables are automatically synchronized between clients and servers of the I/O Driver service by the

VTScada driver module. The communications driver is not responsible for this task.

Rules for Writing a Communications Driver

- Communication drivers are tags and must follow the rules for such.
- For communication driver tags, a variable named "Driver" must be present.

This variable must not be set in the module. The VTScada loader sets it before the module starts.

- Communication driver tags must include a variable named "Ready".

The driver tag code must set this to true when the driver tag is ready to be used.
This variable must not be set to true before:

- The variable Root is set to some object value (usually Self())
- The variable RPCService is set.
- Before Driver\Started becomes true.
- Before Driver\RPCStatus is 2 (server) or 1 (client)

This could be summarized by:

```
Ready = Valid(Root) && Valid(RPCService) && Driver\Started && (Driver-
\RPCStatus == 2 || Driver\RPCStatus == 1)
```

If the driver tag being developed has other constraints they must also be added.

- For tags that do I/O using a communication driver, the driver tag passed in as a parameter must first be converted to an object value.

This object value must be used to launch a copy of any read requests required. This is done with the script code like:

```
If Watch(1, IODevice, Address, ScanRate) &&
           (ValueType(IODevice) == 4 { Text } ||
            Valid(IODevice\Driver) { Driver has started });
[
  IfThen (ValueType(IODevice) == 4 { Text string },
     { Convert text driver name to the object value of }
     { that driver                                     }
    IODevice = Scope(\Code, IODevice)
  );
    { Save a copy of the driver instance for next change }
  Driver = IODevice\Driver;
```

```
    { Delete any previous read requests }
  Driver\DelRead(&RawValue);
    { Start new request for the data }
  Driver\AddRead(Address,
                 1 { # of Elements/Bytes },
                 &RawValue,
                 ScanRate);
]
```

The Address and ScanRate are typically parameters to the tag template module. The Driver variable must be defined in the module.

- Writes to the I/O Device are done by executing a call to the Write module in the driver within a script. The code might look like:

```
Driver\Write(Address, Length { # of elements/bytes }, Data);
```

- A Value variable is common for most tags. For drivers, this value is an error code where 0 indicates no error. Value should be set to the Error from VTSRead/VTSWrite, and must be set on the current primary I/O server.
- All driver tag points must have a variable called Drivers with a class of GROUPS. (not to be confused with "Driver" in point 1).

This should also be a shared variable to conserve RAM. This makes the tag a part of the Drivers group and is necessary to permit the tag to show up in driver tag selection lists and to allow it to set its "Driver" variable before the tags use it.

- Any driver tag that should share the same RPC service for all instances, should define a constant named "RPCService" with the default value being the name of the service.

This name should typically be the name of the driver tag. This will force all instances of the driver tag to share the same server. It will dramatically improve the startup performance of systems that have a large number of instances of the same driver tag type, such as SCADA systems with a large number of RTU's.

An alternative option is possible if the RPCService variable does not have a default value. The driver tag may elect to set this variable based upon some run-time condition (such as a configuration file setting). If the variable exists, it must be set prior to the Driver\Started variable being set in the VTSDriver code.

- If the driver tag uses a serial port, it is a good idea to flush the input buffer before any data is sent. The mechanism that VTScada uses to add and delete read/write commands is such that a read or write that is in progress could be removed from the "queue" in the middle of processing. This means that it is possible that a read or write command could send its data but then be removed from the communications loop before the response comes back. The next read or write command would then pick up the data, if the serial buffer was not emptied. (Refer to the function, SerRcv, for one mechanism to empty the input buffer).
- Do not create a separate module to serve as a low level driver
- Don't declare the variables that are added by diagnostics.
- Add the driver tag to the groups: Trenders, Loggers and Numeric, Drivers.

Related Information:

...Driver Module Instance Object Value

...Error Checking

...Maintaining Statistics

...Common Driver Widgets

...Debugging and Testing Communications Drivers

Driver Module Instance Object Value

The object value of a driver module instance can be used to provide information about how that particular driver is running. This is done by accessing the public variables described in the following table.

Counts	The number of successful communications since starting. Counts is a Long variable.
ErrorCounts	The number of errors encountered since starting. ErrorCounts is a Long variable.
SilentEr-rorCounts	Used only on reads and only by drivers that can have managed sessions or poll for exception, unsolicited data, etc. Indicates that the read could not happen, but this is not necessarily an error. Used to suppress VTSDriver from changing the driver's Value parameter.
Error	The error code number of the current communication attempt. Error is

reset to "0" when the error clears. Error codes follow:

Error	Description
0	No error
1	Not executable in RUN mode
2	Not executable in MONITOR mode
3	Not executable with PROM mounted
4	Address over (data overflow)
B	Not executable in PROGRAM mode
C	Not executable with PROM mounted
D	Not executable in LOCAL mode
10	Parity error
11	Framing error
12	Overrun
13	FCS error (checksum)
14	Format error (parameter length error)
15	Entry number data error (parameter error, data code error, data length error)
16	Instruction not found
18	Frame length error
19	Not executable (due to unexecutable error clear, non-registration of I/O table, etc.)
20	I/O table generation impossible (unrecognized Remote I/O unit, word over, duplication of Optical Transmitting I/O unit)
80	Incomplete response to WRITE
81	Bad serial port parameters
82	Serial port already used
83	FCS error (checksum)
84	No response from PLC (timed-out)

	85	Wrong PLC station address responded
	86	TYPE parameter out-of-range
	87	Garbled/incomplete message
	A0	Aborted due to parity error in transmit data
	A1	Aborted due to framing error in transmit data
	A2	Aborted due to overrun in transmit data
	A4	Aborted due to format error in transmit data
	A5	Aborted due to entry number data error in transmit data
	A8	Aborted due to frame length error in transmit data
	B0	Not executable because the program area is not 16K
LastError	The error code number of the last error encountered by any communications attempt. LastError is not reset to zero when the error clears.	
TimeStamp	The time (in seconds) of the last successful communication since midnight. TimeStamp is a floating-point variable.	
DateStamp	The date of the last successful communication since January 1, 1970. DateStamp is a long variable.	
ErrorTime	The time of the last error encountered in any communication attempt in seconds since midnight. ErrorTime is a floating-point variable.	
ErrorDate	The date of the last error encountered in any communication attempt in days since January 1, 1970. ErrorDate is a Long variable.	
ErrorOwner	An object variable that holds the object value of the read or write statement that encountered the most recent error.	
SCounts [255]	An array of Counts values, one for each Omron PLC.	
SDateStamp [255]	An array of DateStamp values, one for each Omron PLC.	
SError[255]	An array of Error values, one for each Omron PLC.	
SErrorCount [255]	An array of ErrorCounts values, one for each Omron PLC.	

SErrorDate [255]	An array of ErrorDate values, one for each Omron PLC.
SErrorTime [255]	An array of ErrorTime values, one for each Omron PLC.
STimeStamp [255]	An array of TimeStamp values, one for each Omron PLC.
Version	A text variable that is set to the version number of the driver when the driver is started.

For example, if the object value of the driver module was assigned to an object variable called, "Driver" then to display any errors generated by any reads or writes using this driver, use an expression similar to the following example:

```
Output(0,0,1,0,0, Driver\Error,15,0,0,0,0) { Show the current error code }
```

Error Checking

Error checking is an important part of the programming process, as are the testing and debugging processes. There are many possible error situations that must be considered. Error statistics should be examined, and errors recovered from, if possible.

Communication failures are an important issue. They may occur as a result of human error (such as an unplugged cable) or hardware error (such as a power failure or noise in the communication channel). A communication link may not occur on the first attempt, but if the problem is something like distortion in the channel, perhaps after a few retries, a connection may be established. On the other hand, if the problem is a disconnected cable, no number retries will fix the problem. In cases like this, the error should be reported so that the operator can fix it.

When reliable communications are being used, it will be necessary to wait for some type of acknowledgment from the receiver to ensure the message sent was received. In RS-232 serial communications, an echo of data that was received may be sent back, or in Ethernet communications, an acknowledgment (ACK) or a negative acknowledgment (NACK) packet

may be sent to indicate whether or not the packet was received. Similarly, when receiving a packet, it may be necessary to let the transmitter know a message has been received.
The protocol for the packet may use built-in error checking, such as a checksum. If this is the case, the validity of the message can be verified upon receipt, and the message discarded if an error occurred on the communication channel distorting the data.
Another potential issue exists if the driver uses a serial port. The mechanism that VTScada uses to add and delete read/write commands is such that a read or write that is in progress could be removed from the queue in the middle of processing. This means that it is possible that a read or write command could send its data, but then be removed from the communications loop before the response is received. The next read or write command would then pick up the data if the serial buffer was not emptied. Therefore, it is a good idea to flush the input buffer before any data is sent.

Maintaining Statistics

VTScada communication drivers keep at least the following statistics:

LastError	The error code number of the last error encountered by a communications attempt. Unlike the Value variable, LastError is not reset to zero when the error clears.
TimeStamp	The time of the last successful communication, measured in the number of seconds since midnight.
DateStamp	The date of the last successful communication measured in days since January 1, 1970.
ErrorTime	The time the last error occurred, measured in seconds since midnight.
ErrorDate	The date of the last error, measured in days since January 1, 1970.
ErrorOwner	An object variable that holds the object value of the read or write statement that encountered the most recent error.

SCounts, SDateStamp, SError, SErrorCounts, SErrorDate, SErrorTime, STimeStamp	Are arrays of 255 elements, with one element for each possible driver instance. The elements contained in the array correspond to the name of the array.
Version	A text variable that is set to the version number of the driver when the driver is started.

In addition to the statistics that the driver maintains, the VTSRead module maintains statistics for each read attempt that is returned to the VTSDriver module (in VTSDrvr.web). The three statistics that must be returned when VTSRead is finished are:

- **Counts**: The number of successful communications since the driver started.
- **ErrorCounts**: The number of errors since the driver started.
- **Error**: The error code number of the current communication attempt. Error is set to zero when the error clears.

Common Driver Widgets

In order for operators with little or no programming knowledge to see if problems may be occurring with a driver, and acquire some insight into what the problems are, graphics are required for VTScada communication drivers. These graphics can be drawn anywhere on the pages of the VTScada application.

The ShowStats, ShowComm, CommIndicator, SetStats, and ErrorMsg modules are used for graphical display. ShowStats, ShowComm, and CommIndicator are actual graphics modules, while SetStats and ErrorMsg are support modules that provide the statistics and error messages that are displayed. All of these modules can be customized to suit each individual type of I/O device. For example, additional statistics and information can be included in the ShowStats and ShowComm windows; however, the standard displays are generally sufficient.

The ShowStats and ShowComm graphics are buttons labeled with whatever text labels the user specifies, or the defaults: "Show Stats" and "Show Comm". The CommIndicator is a box whose color indicates the status of the communications. The normal color and error color may be chosen when the box is drawn. These buttons can be drawn anywhere on the screen in the VTScada application. All three graphic objects are described in: Drawing Tags).

Debugging and Testing Communications Drivers

There are several tools that may be used for debugging and testing a communication driver, including snooping software, the VTScada Source Debugger (see "Source Debugger"), and simulators. These tools are described briefly here.

Snooping Software

One useful tool for debugging is serial port or TCP/IP port snooping software, such as Stream Team or Ethereal. This software enables the programmer to view all traffic across the computer's serial or TCP/IP port. The programmer may then verify if the information being sent and received meets expectations.

VTScada Source Debugger

VTScada comes with a Source Debugger that is very useful in debugging VTScada code. It enables programmers to watch variables for value changes, insert breakpoints, and view module content and code. Information on the VTScada Source Debugger is provided in "Source Debugger".

I/O Device Simulators

Sometimes the actual I/O device hardware is not available to test with the communication software. On such occasions, it is helpful to write a simulator to emulate the I/O device with which communication is desired. A script application written to use the I/O device's protocol and imitate the I/O device's actions can be run in VTScada at the same time as the driver

tag. This enables testing of the communication driver without having the I/O device prior to installation on site.

Add a New Driver to Your Application

I/O drivers are supplied as separate items in script applications. The drivers for VTScada script applications are delivered as a series of .src files that contain the source code for each driver (see Communication Driver Template for details on driver source files). To use an I/O driver in your script application, you must perform the following steps:

1. Copy the source file (with the .src extension) containing your driver into your application's directory.
2. Modify the application's AppRoot.src file by adding a line of text in the POINTS section similar to the following, where "AmazingDrive" is a fictional device driver:

```
  [ (POINTS)  {================ Modules that are point templates
=========}
    AmazingDrive   Module   "AmazeDrv.src"  { driver for all existent
PLCs};
  ]
```

3. Declare the Config and Common modules in the Plugins section of the AppRoot.src file as follows:

```
[ (PLUGINS) {===== Modules added to other base system modules =====}
   AmazeConfig Module "AmazeCnf.src" { Config };
   AmazeCommon Module "AmazeCmn.src" { Common };
]
```

4. Recompile your application by clicking the Compile button in the VAM.

The driver will now be available for use in the application's tag browser.

Cryptography in VTS

This section provides architectural, programming, and other information on the implementation and use of cryptography in VTScada. It is aimed at both VTScada programmers and advanced engineers performing system configuration, and assumes knowledge of cryptography concepts.

Related Information:

...Cryptography Terms and Abbreviations

...Cryptography Architecture

...Cryptographic Service Providers

...Cryptographic Keys

...Data Encryption and Decryption

...Cryptography Example

Related Functions:

Decrypt	DeriveKey	Encrypt
ExportKey	GetCryptoProvider	GenerateKey
GetKeyParam	ImportKey	SetKeyParam

Cryptography Terms and Abbreviations

BLOB	A generic sequence of bits that contain one or more fixed-length header structures plus context-specific data.
Ciphertext	A message that has been encrypted.
CryptoAPI	An application programming interface that provides services that enable application developers to add cryptography-based security to applications.
Cryptographic key	The session (symmetric) key used during the encryption and decryption processes, and the public and private keys used during the authentication process. Of these three

	keys, the session key and private key must always remain secret.
Cryptography	The art and science of information security. It includes information confidentiality, data integrity, entity authentication, and data origin authentication.
CSP Cryptographic Service Provider	An independent software module that actually performs cryptography algorithms for authentication, encoding, and encryption.
Decryption	The process in which ciphertext is converted to plaintext.
Encryption	The process in which data (plaintext) is translated into something that appears to be random and meaningless (ciphertext). Ciphertext is difficult to unscramble without a secret key.
Key BLOB	BLOB containing an encrypted private key. Key BLOBs provide a way to store keys outside the CSP.
Key container	A part of the key database that contains all the key pairs (exchange and signature key pairs) belonging to a specific user. Each container has a unique name that is used when calling GetCryptoProvider to get a handle to the container.
Plaintext	A message that is not encrypted. Plaintext messages are also referred to as cleartext messages.
Public/private key pair	A set of cryptographic keys used for public-key cryptography.
Public-key algorithm	An asymmetric cipher that uses two keys, one for encryption, the public key, and the other for decryption, the private key. As implied by the key names, the public key used to encode plaintext can be made available to anyone. However, the private key must remain secret. Only the private key can decrypt the ciphertext. The public-key algorithm used in this process is slow (on

	the order of 1,000 times slower than symmetric algorithms), and is typically used to encrypt session keys or digitally sign a message.
Session key	A key used primarily for data encryption and decryption. Session keys are typically used with symmetric encryption algorithms where the same key is used for both encryption and decryption. For this reason, session and symmetric keys usually refer to the same type of key. A session key consists of a random number of approximately 40 to 2000 bits.
Symmetric encryption	Encryption that uses a single key for both encryption and decryption. Symmetric encryption is preferred when encrypting large amounts of data. Some of the more common symmetric encryption algorithms are RC2, RC4, and Data Encryption Standard (DES).
Symmetric key	A single key, typically a session key, used for both encryption and decryption.
Block cipher	A cipher algorithm that encrypts data in discrete units (called blocks), rather than as a continuous stream of bits. The most common block size is 64 bits. For example, DES is a block cipher. Block ciphers are considered more secure than stream ciphers; however, block ciphers tend to execute much slower.
Stream cipher	A cipher that serially encrypts data, one bit at a time.
Initialization vector (IV)	A sequence of random bytes appended to the front of the plaintext before encryption by a block cipher. Adding the initialization vector to the beginning of the plaintext avoids the chance of having the initial ciphertext block the same for any two messages. For example, if messages always start with a common header (a letterhead or "From" line) their initial ciphertext would always be the same, assuming that the same cryp-

tographic algorithm and symmetric key was used. Adding a random initialization vector keeps this from happening.

Related Information:

...Cryptography Terms and Abbreviations

...Cryptography Architecture

...Cryptographic Service Providers

...Cryptographic Keys

...Data Encryption and Decryption

...Cryptography Example

Related Functions:

Decrypt	DeriveKey	Encrypt
ExportKey	GetCryptoProvider	GenerateKey
GetKeyParam	ImportKey	SetKeyParam

Cryptography Architecture

VTScada supports cryptography by means of the Microsoft CryptoAPI (further details on the CryptoAPI may be found with MSDN).
At present VTScada only provides access to a limited portion of the CryptoAPI, but this is sufficient to generate keys to encrypt and decrypt data.
Application developers can use the VTScada cryptography functions without knowing details of the underlying implementation, in much the same way as they can use a graphics library without knowing anything about the particular graphics hardware configuration. The MS CryptoAPI works with a number of cryptographic service providers (CSP) that perform the actual cryptographic functions.
Data encryption transforms a message written in plain text (called "plaintext" in the cryptography community) so that it appears as random gibberish. A good data encryption system makes it difficult to transform

encrypted data back to plaintext without a secret key. The data to be encrypted can be ASCII text, a database file, or any other data you want to store or transmit securely. In this documentation, the term "message" is used to refer to any piece of data; "plaintext" refers to data that has not been encrypted; and, "ciphertext" refers to data that has been encrypted.

Encrypted data can be stored on non-secure media or transmitted over a non-secure network and still remain private. Later, the data can be decrypted into its original form.

Data encryption and decryption are simple processes. When data is encrypted, an encryption key is used. This key is comparable to a physical key that is used to lock a padlock. To decrypt the data, a decryption key is be used. The decryption key is comparable to using a key to unlock a padlock. Encryption and decryption are often done using the same key, but unlike working with physical keys, sometimes encryption and decryption can use different keys from a public/private key pair.

Encryption keys must be kept secret and safe, and must be transmitted securely to other users. This is discussed further in Data Encryption and Decryption. The main challenge is properly restricting access to the decryption key because anyone who possesses it will be able to decrypt all messages that were encrypted with its corresponding encryption key.

Related Information:

...Cryptographic Service Providers

...Cryptographic Keys

...Cryptography Example

Cryptographic Service Providers

The first CryptoAPI function called by an application that uses any cryptographic APIs is the GetCryptoProvider function. This function returns a handle to a particular cryptographic service provider (CSP) that includes

the specification of a particular key container within the CSP. This key container is either a specifically requested key container or it is the default key container for the logged-on user. GetCryptoProvider can also create a new key container.
A cryptographic service provider (CSP) has both a name and a type. For example, the name of one of the CSPs shipped with the operating system is Microsoft Base Cryptographic Provider. It is a PROV_RSA_FULL type provider. The name of each provider is unique; the provider type is not.
When an application calls GetCryptoProvider to obtain a CSP handle, it specifies a provider type and, optionally, a provider name. If both a type and a name are specified, the function loads the CSP with the matching provider type and provider name. The function returns the CSP's handle that provides access to both the CSP and to a key container within the CSP.
When an application calls GetCryptoProvider and specifies a provider type but no provider name, the function looks for a named provider, first checking a list of default named providers associated with the logged-on user and, if that fails, from a list of default named providers associated with the computer. After the provider name has been determined, the GetCryptoProvider function searches for the CSP for that provider, loads it, and returns its handle.

Related Information:

...Cryptography Example

Related Functions:

... GetCryptoProvider

Cryptographic Keys

Cryptographic keys are central to cryptographic operations. They must be kept secret because whoever possesses a given key has access to any

data with which the key is associated. For example, if a key is used to encrypt a file, anyone with a copy of that key can decrypt the file.
There are two types of cryptographic keys: Session Keys and Public/Private Key Pairs.

Session Keys

Session keys, also called symmetric keys, are used with symmetric encryption algorithms. Symmetric algorithms are the most common type of encryption algorithm. They are called symmetric because they use the same key for both encryption and decryption. Session keys are often changed, usually using a different session key for each message encrypted.
Symmetric algorithms are faster than public-key algorithms. Thus, they are preferred when encrypting large amounts of data. Some of the more common symmetric algorithms are RC2, RC4, and the Data Encryption Standard (DES).
Session keys are created by applications using the GenerateKey function. Since a good deal of the activity involving session keys relates to keeping them secret, it is important to keep the number of people who possess a particular session key as small as possible (one or two people is recommended.) These keys are kept internal to the CSP for safekeeping.
Unlike public/private key pairs, session keys are volatile. Applications can save these keys for later use or for transmission to other users by using the ExportKey function to export them from the CSP into application space in the form of an encrypted "key BLOB". The key BLOB may then be imported by another application using the ImportKey function.

Public/Private Key Pairs

Public/private key pairs are used for a more secure method of encryption called asymmetric encryption. Asymmetric encryption is used mainly to encrypt and decrypt session keys and digital signatures. Asymmetric encryption uses public-key encryption algorithms.

Public-key algorithms use two different keys: a public key and a private key. The private key member of the pair must be kept private and secure. The public key, however, can be distributed to anyone who requests it. When one key of a key pair is used to encrypt a message, the other key from that pair is required to decrypt the message. Thus if user A's public key is used to encrypt data, only user A (or someone who has access to user A's private key) can decrypt the data. If user A's private key is used to encrypt a piece of data, only user A's public key will decrypt the data, thus indicating that user A (or someone with access to user A's private key) did the encryption.
Unfortunately, public-key algorithms are very slow, — roughly 1,000 times slower than symmetric algorithms. It is impractical to use them to encrypt large amounts of data. In practice, public-key algorithms are used to encrypt session keys. Symmetric algorithms are used for encryption/decryption of most data.
All keys in CryptoAPI are stored within CSPs. CSPs are also responsible for creating the keys, destroying them, and using them to perform a variety of cryptographic operations.

Related Information:

...Storage and Exchange of Cryptographic Keys

Storage and Exchange of Cryptographic Keys

There are situations where keys must be exported from the secure environment of the cryptographic service provider (CSP) into an application's data space. Keys that have been exported are stored in encrypted key BLOB structures.
There are two specific situations where it is necessary to export keys:

- To save a session key for later use by an application, if, for example, an application has just encrypted a database file to be decrypted at a later time. The application is responsible for storing the encryption key. This is necessary because CSPs do not preserve symmetric keys from session to session.
- To send a key to someone else. This would be easier if the respective CSPs could communicate directly, but they cannot. Because CSPs can't

communicate, the key has to be exported from one CSP, transmitted to the destination application, and then imported into the destination CSP. This process can become more complicated if the communication path is not trusted.

Data Encryption and Decryption

Encryption is the process of translating plain text data (plaintext) into something that appears to be random and meaningless (ciphertext). Decryption is the process of converting ciphertext back to plaintext.
To encrypt more than a small amount of data, symmetric encryption is used. The symmetric key or session key is used during both the encryption and decryption processes. To decrypt a particular piece of ciphertext, the key that was used to encrypt the data must be used. Essentially, a session key consists of a random number, from 40 to 2,000 bits in length. The longer the key, the more difficult it is to decrypt a piece of ciphertext without possessing the key.
The goal of every encryption algorithm is to make it as difficult as possible to decrypt the generated ciphertext without using the key. If a really good encryption algorithm is used, there is no technique significantly better than methodically trying every possible key. Even a key size of just 40 bits works out to just over one trillion possible keys.
It is difficult to determine the quality of an encryption algorithm. Algorithms that look promising sometimes turn out to be very easy to break, given the proper attack. When selecting an encryption algorithm, it is often a good idea to choose one that has been around for a while, and has successfully resisted all attacks.
Data is encrypted using the Encrypt function and decrypted using the Decrypt function. If the data is too big to fit into memory, and can be processed using multiple calls to Encrypt and Decrypt.

Related Information:

...Cryptography Example

Related Functions:

... Encrypt

... Decrypt

Cryptography Example

A handle to a cryptographic service provider (CSP) is obtained using the GetCryptoProvider function. This returns a variable that contains a small wrapper for the CSP handle. The wrapper is necessary to ensure the CSP handle is correctly released when it is no longer referenced.
Calling the GenerateKey function creates a variable that contains a handle to a key. The variable may be copied which generates a duplicate of the key.
Properties of the key may be read and set using the GetKeyParam and SetKeyParam functions respectively.
To transfer secret session keys, the ExportKey and ImportKey functions are used.
Data is encrypted and decrypted using the Encrypt and Decrypt functions.
Example:

```
[
{ Variables and Constants for CSP }
   CSP;
   CSPFail;
   Constant CRYPT_EXPORTABLE = 0x00000001;
   Container = "VTS";
   Constant PROV_DSS_DH = 13;
   Constant CRYPT_NEWKEYSET = 8;
   Constant NTE_BAD_KEYSET = 0x80090016;
{ Items for key generation }
   Key1;
   Key2;
   Constant CALG_DH_EPHEM = 0xAA02;
   Constant KEY_SIZE = 512;
   Constant CRYPT_PREGEN = 0x00000040;
{ Items for key parameter set }
   Constant KP_PERMISSIONS = 6;
{ Items for key parameter get/set }
   KeyP;
   KeyG;
   Constant KP_P = 11 { DSS/Diffie-Hellman P value };
   Constant KP_G = 12 { DSS/Diffie-Hellman G value };
```

```
   Constant KP_Q = 13 { DSS Q value };
   Constant KP_X = 14 { Diffie-Hellman X value };
   Constant KP_Y = 15 { Y value };
{ Items for export/import }
   PubKey1;
   PubKey2;
   Constant PUBLICKEYBLOB = 0x6;
   Key3;
   Key4;
{ Algorithm conversion }
   Constant KP_ALGID = 7;
   Constant CALG_RC4 = 0x6801;
{ Variables for encryption / decryption }
   PlainText1 = "abcdefghijklmnopqrstuvwxyz0123456789";
   CipherText1;
   PlainText2;
]
Init [
   If 1 Main;
   [
      { Get the CSP }
      CSP = GetCryptoProvider(PROV_DSS_DH, Invalid, Container,
Invalid, CSPFail);
      IfThen(CSPFail == NTE_BAD_KEYSET,
      { Not used this container before, make a new one }
         CSP = GetCryptoProvider(PROV_DSS_DH, Invalid, Container,
CRYPT_NEWKEYSET, CSPFail);
      );

      { Make a key }
      Key1 = GenerateKey(CSP, CALG_DH_EPHEM, KEY_SIZE << 16 || CRYPT_
EXPORTABLE);

      { Get the key parameters }
      KeyG = GetKeyParam(Key1, KP_G);
      KeyP = GetKeyParam(Key1, KP_P);

      { Make another key using the parameters }
      Key2 = GenerateKey(CSP, CALG_DH_EPHEM, (KEY_SIZE << 16) ||
CRYPT_PREGEN);
      SetKeyParam(Key2, KP_G, KeyG);
      SetKeyParam(Key2, KP_P, KeyP);
      SetKeyParam(Key2, KP_X);

      { Export the public keys from both keys, and import to each
other }
      PubKey1 = ExportKey(Key1, PUBLICKEYBLOB);
      PubKey2 = ExportKey(Key2, PUBLICKEYBLOB);
      Key3 = ImportKey(CSP, PUBLICKEYBLOB, PubKey1, Key2);
      Key4 = ImportKey(CSP, PUBLICKEYBLOB, PubKey2, Key1);

      { Now convert the shared secret key to the bulk encryption key
}
      SetKeyParam(Key3, KP_ALGID, CALG_RC4);
      SetKeyParam(Key4, KP_ALGID, CALG_RC4);

      { Now use the keys to encrypt and decrypt some data }
```

```
      CipherText1 = Encrypt(Key3, PlainText1, 1, 0, 0);
      PlainText2 = Decrypt(Key4, CipherText1, 1, 0, 0);
   ]
]
```

The sample code demonstrates instantiating a CSP, generating two public / private key pairs, exporting the public parts of the keys, importing the public keys to create two new keys that have a shared secret, converting the resulting keys to a symmetric key and encrypting and decrypting data. For the sake of clarity, error handling is not shown.
Some points to note are:

- Many parameters to the Cryptography functions and statements are constants that are defined in WinCrypt. The functions and statements do not interpret these values, but pass them directly to the CryptoAPI. This enables all the current and any new features of the CryptoAPI to be used without having to modify engine code if the parameters were directly interpreted.
- The GetCryptoProvider function returns a value that represents a handle to the required CSP.
- As the CSP isn't named, the actual CSP returned will depend on the OS and version of Internet Explorer. For this reason it's recommended that key sizes and other variable parameters are explicitly set.
- GenerateKey is a function that returns a value holds a handle to the generated Key.
- GetKeyParam and SetKeyParam are used to read and write to parameters of a Key.
- The public part of Key1 is exported to a text variable that is returned by ExportKey, which is then imported into Key2 producing Key3 using ImportKey. A similar operation is performed to produce Key4. Key3 and Key4 now contain a shared secret.
- The keys containing the shared secret are converted to a symmetric key using SetKeyParam.
- Encrypt returns a text string that is the ciphertext of the plaintext string passed as a parameter. Decrypt returns a text string that is the plaintext of the ciphertext string passed as a parameter.
- A variable containing a CSP handle has a type of 36 and if printed will display the textual name of the CSP.

- A variable containing a Key handle has a type of 37 and if printed will display the hexadecimal value of the algorithm ID of the key.

Related Information:

...Cryptography Terms and Abbreviations

...Cryptography Architecture

...Cryptographic Service Providers

...Cryptographic Keys

...Data Encryption and Decryption

Related Functions:

Decrypt	DeriveKey	Encrypt
ExportKey	GetCryptoProvider	GenerateKey
GetKeyParam	ImportKey	SetKeyParam

Custom Tag Types

The cornerstone of every VTScada application is a group of components called "tags" (sometimes referred to as "points"). These represent the various equipment processes that make up your system, and enable you to create a chain of communications from VTScada to your physical equipment. A typical application based on the VTScada layer will include a set of tag types, with which you can create as many tag instances as your license allows.

You can extend VTScada's feature set by creating new kinds of tag. Some examples include:

- A tag that collects data from several inputs, generating a combined value.
- A driver tag for a new I/O device.
- A controller that starts each motor in a group several seconds apart, thereby avoiding load spikes.

There are three ways to create new types of tag:

- Create a Context type, adding properties and child tags that fully describe a machine, a site, an object or process.
- Write a new type entirely from scratch using the information within this chapter.
- Do both of the above, starting with a Context tag to define the fundamental structure, then extending its source code to add completely new features.

It is rare for anyone to write a new type entirely from scratch. If a Context-based structure does not fulfill the need, then it makes an excellent starting point for further development using custom code.

This chapter focuses only on standard tags. If your intent is to create a Communication Driver, you should first learn the material here before attempting to extend the tag with the features required for a driver.

Note: Every tag has three names: "Shortname" is the name of just that tag, alone. "Name" is the full name including all parents in the hierarchy. "UniqueID" is the guaranteed unique identifier, belonging to that one tag instance alone.

For any purpose that requires a lasting connection to a tag, such as

---

alarm state & history, network values, communication with other machines, you should always use the unique ID rather than the name. TagObj.UniqueID (or just UniqueID from within a tag module) can be used to get the UniqueID for a tag.

Guide to This Chapter

The first topics of this chapter introduce the most basic possible tag structure. Enhancements such as data logging, alarms, etc. will be described and added to the sample code individually.
At the end of each main topic, a list of rules is provided. These will summarize the information presented in that topic and may be used as a checklist when creating your tags.
The topics are as follows:

- Tag Basics
- Module structure including parameters, required and common variables.
- Standard submodule declarations.
- State code for tags.
- The Refresh module.
- Tag Configuration Folders
- Declaring the module.
- State code for configuration modules.
- How to switch tabs.
- How to display the data input fields.
- Adding expression support for parameters.
- Drawing Tags
- Selecting the VTScada widgets that will be available to your tag.
- Creating your own widgets.
- Detecting run mode versus edit mode and responding accordingly.
- Creating a properties dialog panel for configuring your drawing object.
- Indicating questionable and manual data.

- Responding to user actions
- The Common module.
- Display a tool tip.
- Display a right-click context menu.
- Display a trend window.
- Linking to driver I/O
- Reading from an I/O driver.
- Writing to an I/O driver.
- Logging tag data
- Configuring so that a logger can be attached.
- Build logging into your tag.
- Alarms
- Configuring so that an alarm tag can be attached.
- Build alarm features into your tag.
- Containers and Contributors
- Contribute information from your tag to a container.
- Collect information from contributors.

Terms for Tag Types

The following terms and abbreviations are used throughout this chapter.

Tag Template Module	A module that defines the structure of a tag. An example of a typical tag module is "AnalogInput". The tag module has formal parameters that correspond one-for-one with the fields in the tag properties database.
Tag	For the purposes of this section, a tag is defined as a named instance of a Tag Template Module. For example, an instance of the DigitalInput template is a digital input tag. Another definition to describe a tag is, "A software component that can communicate with objects in the outside world. A tag can be used to accept input or generate output.
Database	The Database module is at the highest scope in a VTScada application, and is the location where all instances of the tags for the application are defined.

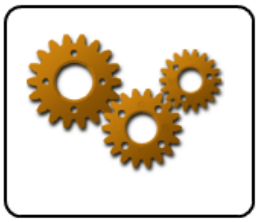
Library	The Library module is the location in a VTScada application where most of the modules and non-tag related variables are defined. The Library module is at the scope level just under the Database module. This module's instance goes by the name of "\VTSDB" when referenced within a VTScada application.
Point	An older term for what is now called a "tag".
Tag Properties Folder	Also referred to as a "tag configuration folder" or a "config folder". A dialog with a series of tabs, each of which contains a set of tag properties of a specific category and is appropriately labeled according to the property fields it contains (e.g. the ID tab contains properties that identify the tag.) The tag properties folder is a convenient tool that enables users to enter data into the Tag Properties Database.

Tag Template Modules

Tag template modules contain the code that defines the tag types available to your application. Tag template modules define not only what data types are available, but also how those data types can be displayed, their logic, their control actions, their alarm behavior, and all other characteristics native to tags of that tag type.

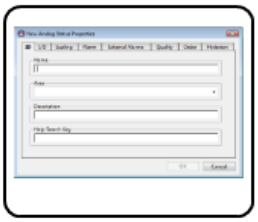
The modules for a tag template will typically be stored in three separate files. This makes it easier to maintain and update the tag's code and makes it easier to find the various parts of the complete tag template. In the following example "Cnf" (appended to the tag name) indicates "Configuration modules" and "Cmn" indicates "Common modules"

TagName.src



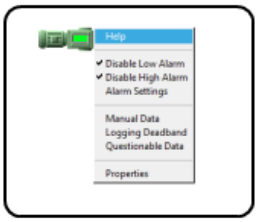
The tag's structure and the code that makes it work.

TagNameCnf.src



Displays the tag's configuration folder and displays & accepts configuration information

TagNameCmn.src



Displays and provides the functionality for the tag's right-click menu. Every tag drawing method will call this module for the associated tag type in response to a right-click.

All of the custom tags that you add to an application must be declared in the [POINTS] class of the application's AppRoot.SRC file. The configuration module and common module may be submodules in the tag's source file, but it is better to keep them as separate files and declare these in the [PLUGINS] class of AppRoot.SRC.

The Basic Tag - TagName.SRC

A tag is built using VTScada **module**[1] and **state**[2] code, just like any other piece of VTScada code. If you are not familiar with the VTScada language, please study the chapter, The VTScada API, before continuing in this chapter.
Like all modules, the tag template will include:

- A parameter section
- Declaration of variables and submodules used by the tag template module.
- An initialization state, where start-up tasks are performed and the tag's Refresh module is launched.
- The tag's main state.

Related information that you may need:

...Tag Configuration Parameters

...The Tag Variables Section

...Rules for Tag Variables, Constants and Modules

---

[1]A collection of states, scripts, variables, parameters, comments and possibly other modules, all of which make up a VTS program. Modules are separate tasks that run simultaneously in an application. Behind every page is a module and behind every tag on that page is an instance of another module (usually with submodules controlling separate tasks).
[2]A collection of statements, grouped together within square brackets and given a name. A state is the part of the module that performs a task. Modules can have many states, but only one may be active at a time.

Tag Configuration Parameters

A tag template module must be configured with a set of formal parameters that will correspond one-for-one to the fields of a table in the tag properties database. Each tag module begins by defining the tag's parameters, within round parenthesis. For example, the Analog Status tag begins as follows:

```
{========================= AnalogStatus
================================}
(
  Name          <:TagField("SQL_VARCHAR(255)", "Name",          0 ):>
                        { Point Name
};
  Area          <:TagField("SQL_VARCHAR(255)", "Area",          1 ):>
                        { Point area
};
  Description   <:TagField("SQL_VARCHAR(255)", "Description",   2 ):>
                        { Point Description
};
```

Many more parameters follow those shown here.
For each parameter, a parameter constant definition must also be assigned. This will be discussed in The Tag Variables Section.
The first three parameters, Name, Area and Description are mandatory and must be as shown in this example. The parameter, Name, must be the first parameter in the list. The others may be entered in either order, but it is recommended that the standard order of Name, Area and Description be maintained.
You may define more parameters as needed, using the following format:

```
DeviceTag <:TagField("SQL_VARCHAR(255)", "I/O Device Name", 3, FALSE
{ Encrypt }, "SitePoint", "IODeviceLabel" ):>
```

where
"DeviceTag" is a parameter definition.
<: :> is the MetaData operator (This marks the mandatory Tag Parameter Metadata section)
TagField is a structure with the following data:

- The SQL data type in the database. The data type should match a list of standard names (see list in the following topic). The configuration settings contain a block translating the standard data type names to the actual data type

names used by your ODBC compliant database program. This field is mandatory.
Note that the data type can be defaulted to SQL_VARCHAR(255). This type must be used for the 'Name' parameter. All other fields can use SQL_VARCHAR(255) or SQL_LONGVARCHAR. SQL_LONGVARCHAR is preferred since all other fields can be expressions

- The name of the field to be created in the database. This is required only if the field name will differ from the parameter name. If not provided, then the field name in the database will be the same as the parameter definition.
- The column number of the field *if required for a legacy application*. This parameter is only required if the application is to be used in versions of VTS prior to 8.0. Defaults to the first unused index number, starting from 0.
- (Optional) A Boolean that if TRUE, specifies that any value stored in this parameter should be encrypted.
- (Optional) Avatar. If the parameter resolves to an object, this names the variable where that object is stored.
  In practice, this is used only in I/O tags for the I/O device parameter, and then only if that type will be used to trigger alarms. In this specific case, the avatar should be set to "SitePoint".
- (Optional) Moniker. Label for the Avatar parameter, used for the graphical user interface. Typically, "IODeviceLabel", when the avatar is "SitePoint".

Note that Field Names may have spaces, but Parameter Names cannot. You may also assign a default value for any parameter by adding " = SomeValue" after the closing angle bracket. For example, the declarations for the Analog Status tag's Unscaled Max and the Units parameters are declared as follows:

```
UnscaledMax    <:TagField("SQL_DOUBLE",      "Unscaled Max", 7 ):> =
4095;
Units          <:TagField("SQL_VARCHAR(50)", "Units",        10):> =
"%";
```

In some cases, the default value should be a particular type of parent tag or a defined VTScada tag. For example:

```
DeviceTag     <:TagField("SQL_VARCHAR(255)", "I/O Device Name", 3,
FALSE, "SitePoint", "IODeviceLabel"):> = "*Driver";
HistorianName <:TagField("SQL_VARCHAR(255)"):> = #SYSTEM_HISTORIAN
```

Related information that you may need:

...SQL Data Types for Tag Parameters

...Adding New Parameters to Existing Tags

...Example – The Analog Status Tag's Parameters

...Rules for Tag Variables, Constants and Modules

SQL Data Types for Tag Parameters

In practice, only a few SQL data types are used. The most common are:
SQL_VARCHAR(255) – for most text. In this example, 255 characters are being allocated. Must be used for the Name parameter. May be used for all other parameters, although SQL_LONGVARCHAR is preferred.
SQL_LONGVARCHAR – Preferred for all fields other than Name, since this allows space for expressions.
The full list of available SQL data types. Note that SQL_LONGVARCHAR is preferred in all cases except the Name parameter.

- SQL_BIT
- SQL_TINYINT
- SQL_BIGINT
- SQL_LONGVARBINARY
- SQL_VARBINARY
- SQL_BINARY
- SQL_LONGVARCHAR
- SQL_UNKNOWN_TYPE
- SQL_CHAR
- SQL_NUMERIC
- SQL_DECIMAL
- SQL_INTEGER
- SQL_SMALLINT
- SQL_FLOAT
- SQL_REAL
- SQL_DOUBLE
- SQL_DATE

- SQL_TIME
- SQL_TIMESTAMP
- SQL_VARCHAR

Adding New Parameters to Existing Tags

You may add a new parameter to an existing tag by adding it to the end of the parameters section. Upon compiling the application and creating instances of the tag type, the new field will be added to the database. Note that, to use the new parameter you must also assign a parameter constant definition, as described in The Tag Variables Section.

Rules for Parameters

Summarizing the information in the preceding topics:

- The names of the first three parameters for all tag template modules must always be: "Name," "Area," and "Description".
- An unlimited number of tag parameters may be defined. It should be noted that child tags have a 256 parameter limit.
- Do not use the period character (.) in the names of any of the fields in the table corresponding to your tag type template (in the tag properties database). Periods are converted to a number hatch, and will prevent data from being written to the database.
- Do not use SQL reserved words as parameter names.

Encrypted Parameters

You may designate that the value of a parameter, as stored in the tag file, shall be encrypted. This does not encrypt the parameter in any VTScada user interface element. The purpose is to block unauthorized inquiries on the part of persons who do not have access to the application, but do have access to your server.

Encrypted parameters can only be defined for your custom tags. They are not used in any VTScada tag. Note that encryption is not applied retroactively if you modify your tag template file. Only tags created after the encryption flag has been set will have encrypted parameters.

In the parameter definition, the encryption flag is set as a Boolean in the TagField definition. In the following example, the parameter, "Illegible," is set to be encrypted:

```
(
   Name <:TagField("SQL_VARCHAR(255)"):>;
   Area <:TagField("SQL_LONGVARCHAR"):>;
   Description <:TagField("SQL_LONGVARCHAR"):>;
   Illegible <:TagField("SQL_LONGVARCHAR", "", Invalid, TRUE):>;
   HelpKey <:TagField("SQL_LONGVARCHAR"):>;
)
```

Within VTS, the value of "Illegible" will be completely visible. But, in the tag file, the it will not. For example, the value, "The quick brown fox...," will be stored as ":ò7½;j¢×É](Tù†".
The three parameters following the data type are:

- "" – the name to create in the database. Blank as this does not differ from the first field.
- The column number of the field. Invalid as this is not required.
- The Boolean designating that any value stored in this parameter should be encrypted.

Example – The Analog Status Tag's Parameters

As an example, the complete list of parameters for the Analog Status tag is provided. If your tags will include similar parameters, it is recommended that you use similar names and SQL data types. Note the lack of column numbers for parameters added since the release of VTS 8.0. Since older applications will not have these parameters, the column number was not required for backward compatibility.

```
(
 Name               <:TagField("SQL_VARCHAR(255)", "Name", 0 ):>
    { Point Name };
 Area               <:TagField("SQL_VARCHAR(255)", "Area",         1
):>
    { Point area };
 Description        <:TagField("SQL_VARCHAR(255)", "Description",  2
):>
    { Point Description };
 DeviceTag          <:TagField("SQL_VARCHAR(255)", "I/O Device Name",
3 , FALSE { Encrypt }, "IODevice" IODeviceLabel"):> = "*Driver"
    { Site Point driver };
 Address            <:TagField("SQL_VARCHAR(255)", "Address",      4
```

```
):>
    { Address for this input };
 ScanRate           <:TagField("SQL_DOUBLE", "Scan Rate",          5
):> = 1
    { Rate at which to scan (fastscan only) };
 UnscaledMin        <:TagField("SQL_DOUBLE", "Unscaled Min",       6
):> = 0
    { Minimum Unscaling counts };
 UnscaledMax        <:TagField("SQL_DOUBLE", "Unscaled Max",       7
):> = 4095
    { Maximum Unscaling counts };
 ScaledMin          <:TagField("SQL_DOUBLE", "Scaled Min",         8
):> = 0
    { Minimum scaling value };
 ScaledMax          <:TagField("SQL_DOUBLE", "Scaled Max",         9
):> = 100
    { Maximum scaling value };
 Units              <:TagField("SQL_VARCHAR(50)", "Units",
10):> = "%"
    { Engineering units };
 AlarmLo            <:TagField("SQL_LONGVARCHAR", "Alarm Lo",
11):> = 0
    { Low Alarm Setpoint };
 AlarmHi            <:TagField("SQL_LONGVARCHAR", "Alarm Hi",
12):> = 100
    { High Alarm Setpoint };
 PriorityLo         <:TagField("SQL_DOUBLE", "Priority Lo",
13):>
    { Low Alarm priority };
 PriorityHi         <:TagField("SQL_DOUBLE", "Priority Hi",
14):>
    { High Alarm priority };
 InhibitLo          <:TagField("SQL_LONGVARCHAR", "Inhibit Lo",
15):> = 1
    { Set to inhibit the low alarm };
 InhibitHi          <:TagField("SQL_LONGVARCHAR", "Inhibit Hi",
16):> = 1
    { Set to inhibit the High alarm };
 AlarmSound         <:TagField("SQL_VARCHAR(255)", "Sound",
17):>
    { Name of a .WAV file to play when alarm set };
 ManualValue        <:TagField("SQL_VARCHAR(255)", "Manual Value",
18):>
    { Manual value for Point };
 Threshold          <:TagField("SQL_DOUBLE", "Deadband",
19):>
    { Deadband value for logging changes };
 Questionable       <:TagField("SQL_DOUBLE", "Questionable Data",
20):> = 1
    { Set to true when data is questionable };
 Quality            <:TagField("SQL_VARCHAR(255)", "Data Quality",
21):>
    { Point/value to determine quality of this point };
 DisplayOrder       <:TagField("SQL_DOUBLE", "DisplayOrder",
22):> = 0
    { Order to Display this point };
 HelpKey            <:TagField("SQL_VARCHAR(255)", "Help Key",
```

```
23):>
    { Index into help system };
 PopupLo            <:TagField("SQL_DOUBLE", "Popup Lo"
):> = 0
    { Set to enable a popup for the low alarm };
 PopupHi            <:TagField("SQL_DOUBLE", "Popup Hi"
):> = 0
    { Set to enable a popup for the high alarm };
 AlarmLoDeadband    <:TagField("SQL_LONGVARCHAR"
):> = 0
    { Deadband for Lo Alarm };
 AlarmHiDeadband    <:TagField("SQL_LONGVARCHAR"
):> = 0
    { Deadband for Hi Alarm };
 AlarmLoDelay       <:TagField("SQL_LONGVARCHAR"
):> = 0
    { Delay for Lo Alarm };
 AlarmHiDelay       <:TagField("SQL_LONGVARCHAR"
):> = 0
    { Delay for Hi Alarm };
 AlarmLoRearmTime   <:TagField("SQL_LONGVARCHAR", "Lo Rearm Time"
):> = 3600
    { Time in seconds before acked alarm rearms };
 AlarmHiRearmTime   <:TagField("SQL_LONGVARCHAR", "High Rearm Time"
):> = 3600
    { Time in seconds before acked alarm rearms };
 AlarmLoRearmEnable <:TagField("SQL_DOUBLE", "Lo Rearm Enable"
):> = 0
    { Flag, TRUE if acked alarms to be rearmed };
 AlarmHiRearmEnable <:TagField("SQL_DOUBLE", "High Rearm Enable"
):> = 0
    { Flag, TRUE if acked alarms to be rearmed };
 HistorianName      <:TagField("SQL_VARCHAR(255)"
):> = #SYSTEM_HISTORIAN
    { Historian Tag name };
 EnableOutput       <:TagField("SQL_BIT" ):> = 0
    { Set to enable data to be written to the IODevice as well as
read from the IO. };
 SecurityBit        <:TagField("SQL_VARCHAR(255)", "Security Bit",
):>
    { The bit number in the security manager which enables control.
};
 StyleTag           <:TagField("SQL_VARCHAR(255)" ):> = "*Style Set-
tings"
    { Style settings for drawing methods};
 EnableLogging      <:TagField("SQL_LONGVARCHAR" ):> = TRUE
    { Enables logging, default is TRUE, can be a constant, tag value
or expression };
 RangeMin           <:TagField("SQL_DOUBLE" ):>
    { Minimum range value };
 RangeMax           <:TagField("SQL_DOUBLE" ):>
    { Maximum range value };
)
```

The Tag Variables Section

The tag variables section follows the parameters and is enclosed in square brackets [ ].
This section declares (and in some cases, initializes) local variables, constants, and modules related to the tag. The following is a list of some of the items that may be found here:

- Required variables, such as Root and Value
- Other local variables such as RawValue, RawTS, DisplayAddress, and SitePoint
- Module declarations such as Refresh, NewData and Alarms
- Plugin declarations such as the ConfigFolder and Common modules
- Graphics declarations for VTScada widgets that will be available to this tag.
- Group membership declarations such as "Numeric" and "Loggers".
- Constant declarations including parameter constant definitions, HelpID values, and NumTagFiles.

These items will be described in later topics of this chapter.

Related information that you may need:

...Rules for Tag Variables, Constants and Modules

...Required Variables

...Constant Definitions

...Other Constants

...Submodule Declarations

Required Variables

If instances of your tag are to have value, whether for use in widgets or logging, there must be a variable named "Value". The class type (indicating the data type) of Value must also be declared. Note that this is a type declaration, not a value initialization.

```
    { Variables }
[
  Value (5)                { Scaled value for this point.        };
]
```

The possible classes for Value are:

- Class 1 – Bit
- Class 2 – Unsigned byte
- Class 3 – 16-bit integer
- Class 4 – 32-bit integer
- Class 5 – Double precision floating point
- Class 6 – Text

Note: The value of this variable must be set in a script, not in steady-state.

If the tag template module provides for a manual value or an external value then care must be taken to use that value whenever it is set. An example of code showing a manual value being used can be seen in the topic, The Refresh Module.
Another variable that must be part of every tag template is "Root". Root will be used to identify the individual instances of the module. This is required to allow the parameter editing tools to access the parameters of the tag. It is not assigned a class type.

```
    { Variables }
[
  Value (5)              { Scaled value for this point.          };
  Root                   { Set to individual instance of this module
};
]
```

Optional Variables

In addition to the required variables, there are several others that will be found in many tags. These include the Raw Value as read from I/O, a local object value for the I/O device driver, etc.
Of note is "DisplayAddress". If present, the Tag Browser is able to display the tag's configured I/O address. Since new types of tags have added to VTScada over many years, there is some variation in the name used for the I/O address field. By creating a variable with the name, Dis-playAddress, and ensuring that it is always holds the current value of the I/O address, from whatever parameter stores that value for your tag, you

can ensure that instances of your tag type display their I/O address in the Tag Browser.
As an example, a partial list of variables from the Analog Status tag is provided:

```
{ Variables }
  RawValue          { Data value read from the IO
};
  RawTS             { UTC Timestamp from NewData
};
  Value (5)         { Scaled value for this point.
};
  SitePoint         { Object value of DeviceTag
};
  OldSitePoint      { Previous value of SitePoint
};
  Style             { Object value of the StyleSettings tag
};
  Started = 0       { Let drawing methods know that Value has been
restored. };
  DisplayAddress    { Address for display in the Tag Browser
};
  QualityIssue      { TRUE if there is a data quality issue
};
```

The SitePoint variable is required only if an Avatar property was defined in the DeviceTag parameter. If so, then in the Main state of the tag, this property must be set to the object value of the device:

```
SitePoint = Scope(Root, DeviceTag);
```

More variables will be required for local calculations, logging, built-in alarms, and other purposes.

Constant Definitions

A numbered constant must be assigned for each of the declared parameters. These will be in the same order that the parameters were declared. For example, some of the parameter constants for the Analog Status tag are as follows:

```
   { Parameter constant definitions }
  Constant #Name                = 0;
  Constant #Area                = 1;
  Constant #Description         = 2;
  Constant #SitePoint           = 3;   { link to the I/O device }
  Constant #Address             = 4;   { the I/O address        }
  Constant #ScanRate            = 5;
  Constant #UnscaledMin         = 6;
  Constant #UnscaledMax         = 7;
```

```
Constant #ScaledMin          = 8;
Constant #ScaledMax          = 9;
Constant #Units              = 10;
Constant #AlarmLo            = 11;
Constant #AlarmHi            = 12;
```

Other Constants

NumTagFiles: In addition to the constant definitions for the parameters, you should also include one for NumTagFiles.

```
Constant NumTagFiles = 256;
```

This value determines the number of tag files that are used to store all of the instances of this particular type. For example, the tag instances for the Analog Status tag type are randomly distributed between 256 files. There is a trade-off between application start-up performance (faster with fewer tag files) and online tag editing performance (faster with fewer tags per file, hence faster with more tag files). In general, you will get good online-editing performance with up to 1000 tags/file. Therefore, since NumTagFiles in the Analog Status tag is set at 256 files, an application should have good online editing performance with up to 256,000 of these tags.

Note: After the first instance of a tag has been created, any and all changes to NumTagFiles will be ignored.

The default if NumTagFiles is not defined for a tag, is 64. This will be provide good performance for up to 64,000 instances of that tag type in an application.

PriorityLoad: Tags may be given a constant named "PriorityLoad". Tags that contain this constant, set to a value of 1, will be started before other tag types.

PriorityReady: If a tag module has a PriorityLoad variable, it may optionally have a variable named PriorityReady as well. If PriorityReady is used, the tag loading code will wait until its value is set TRUE (non-zero) before starting any non-priority tag types.

Note: Use PriorityReady with caution. Failure to ensure that its value is set TRUE will cause tag-loading to stop.

ContextType: Of particular note is the constant, ContextType. While not required, this is extremely useful when your tag definition is used as part of a parent-child tag structure. The context type declaration informs child tags of what type this is. For example, Alarm tags will automatically connect to the first *Numeric parent, Analog Input tags look for a parent of context type *Driver.

By specifying the context type of your tag, you make it possible to link your tag to the parent-child hierarchy, automatically. If your tag does not specify its ContextType explicitly, one will be created automatically, using the name of the tag.

Child tags are told what ContextType to use for a parameter by providing that value in the parameter definition. For example, in the Deadband tag's parameter list, you will find the following parameter declaration:

```
MonitoredValue    <:TagField("SQL_LONGVARCHAR",  "MonitoredValue",
3 ):> = "*Numeric";
```

In the list of constants for an Analog Status, Calculation, Digital Input, or other tag, you will find:

```
  Constant ContextType = "*Numeric";
```

BuiltInAlarm: Include and set true if the tag browser is to inform developers that this tag includes one or more built-in alarms.

```
Constant BuiltInAlarm = TRUE { Flag - TRUE if this tag has a built in
alarm };
```

Assigning Tag Groups

Tag groups are collections of tag types that share some logical relationship. One example of a tag group is the Drivers group, which lists all of the I/O drivers in the system.

A tag may be a member of multiple tag groups. For example, the analog input tag type belongs both to the Analogs group (as it has an analog value), as well as to the Numeric group (as it has a numeric value).

Groups declaration of the Analog Input tag:

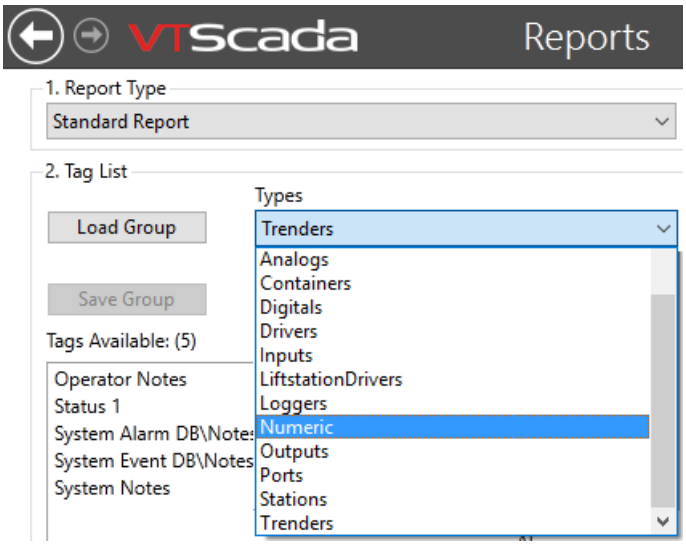
```
[ (GROUPS)
   Shared Numeric;
   Shared Analogs;
```

```
   Shared Trenders;
]
```

Groups declaration of the Context tag:

```
[ (GROUPS)
   Shared Container { We are a container tag };
]
```

One purpose of tag groups is to allow the PSelectObject module to display a drop-down list of all the tags that belong to a group, rather than all of the tags of a particular tag type. Again, drivers are an excellent example: the ConfigFolder module requires a list of all driver tags configured for the application so that it can display them in the I/O device drop-down list on the I/O tab for the end user to choose from.



Duplicate Tag Groups

You may create your own tag group module within an OEM layer, or within an application directory. If doing so, be careful not to provide the same name as an existing group module. If you do, then the tags belonging to the existing group are merged with the tags belonging to the new group, creating a combined list, rather than overwriting either list.

AppRoot.src GROUPS Section

To create a tag group, you must use the (GROUPS) section within the application's AppRoot.src file (see AppRoot.src Root File for a Standard Application). Any identified modules within this section will be searched

for any variables that are of class POINTS. These variable names are used as the names of the tag modules to include in the group.
Additionally, the tag module itself can define a GROUPS class variable that should be the name of the group in which this tag should be included. If the group doesn't already exist, the VTScada code creates it at load time. These variables should be declared as Shared to conserve RAM (see "Shared Variables").

DrawLabel Variable

There should be a class 0 variable called "DrawLabel" in the tag group module. The default value of this variable is used as the name of the variable in the application's configuration that contains the text descriptor for the group. This description is placed in the configuration variables so it can easily be translated into different languages, or otherwise modified without requiring you to directly modify the application's code. If the "DrawLabel" variable is not present, the tag group module's name is used as its description. This description appears in the Tag Browser.

Submodule Declarations

Modules that the tag will use must be declared in the variables section. This includes submodules of the tag and external modules that the tag will call, such as the widgets, alarm methods, logging, etc. Two of particular note are "Refresh" and "Common" – modules that must be implemented in every tag template.
The Refresh subroutine will be launched by the tag's initialization state and will be called every time that the tag's data changes. The declaration will appear as follows:

```
  Refresh         Module        { Called when point changes
}
```

Referring to the following diagram, you may expect to see declarations for TagNameCmn and TagNameCnf. In the Analog Status tag, these are declared in the [Plugins] class as follows:

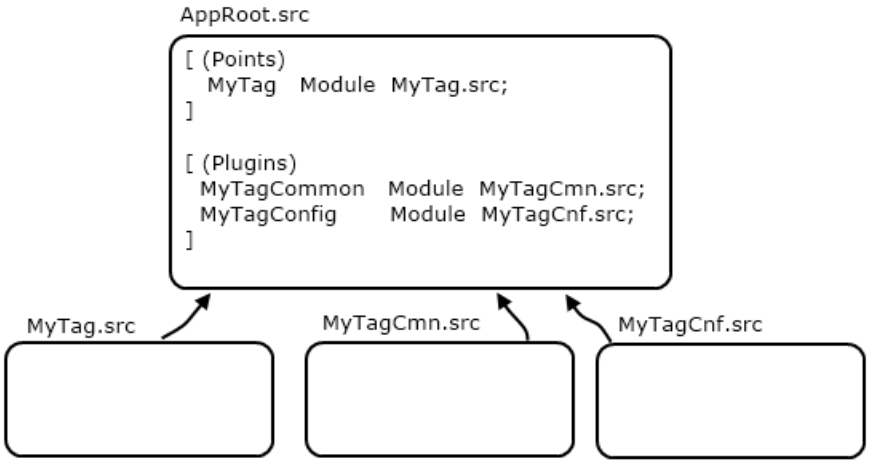
```
  [ (PLUGINS)
    Shared ConfigFolder        = "AnalogStatusConfig";
```

```
  Shared Common              = "AnalogStatusCommon";
]
```

(While it is common to declare these modules as Plugins, other techniques are possible. In a smaller tag, it may make sense for these to be submodules within the tag's source file.)

You might notice in this example, that the name "AnalogStatusConfig" does not match "AnalogStatusCnf" and also that the source file for the module is not provided. How then does the tag find the correct module? The answer is that, in addition to the tag itself, both the Config and the Common modules are declared in AppRoot.src. For your custom tags, you will add the declarations to AppRoot.src in your application or OEM layer. For example (taken from VTScada layer's AppRoot.src):

```
[ (POINTS) {===== Modules which are point templates =====}
  AnalogStatus      Module "AnalogStatus.SRC";
]

[ (PLUGINS)
  AnalogStatusConfig       Module "AnalogStatusCnf.SRC";
  AnalogStatusCommon       Module "AnalogStatusCmn.SRC";
]
```



Related Information:

The Refresh Module

Rules for Tag Variables, Constants and Modules

- A variable named, "Value" is required for most tags. It is usually declared to be of Class 1 through Class 6.

- Any tag template that has a value (such as an input tag) must name this variable "Value". This variable must be set in a script. The setting of the value might use code similar to that shown here, as taken from a digital input tag:

```
If Watch(0, Bit1, Bit2) &&  ! ExternalValue;
[
   Value = 2 * Bit1 + Bit0;
]
```

- The "ExternalValue" variable must be defined in a tag template module with a default value of 0 for tags that have a value. The ExternalValue variable must be used to prevent setting of the Value variable when true. This exists for specialized applications that might provide a value for the ExternalValue variable.
- A variable named, "Root" must be defined in the tag template module, and must be set to the instance of the module. The following displays the correct syntax.

```
Root = Self();
```

  The Root variable enables access to the instance value of the tag by the tag's configuration modules.

Tag States

As with all VTScada modules, the work is done with state code. The first state in a module will always run automatically. In the case of a tag, this state has three tasks: Set the variable Root to "Self", launch the Refresh module, and transfer control to the main module.
While it is common practice to name a module's initialization state, "Init" and the main state "Main", you should instead use distinctive names for these states in your tags. It will be easier to for you to debug your code if the state names reflect the tag modules they are a part of.
Reducing the Analog Status tag's initialization state to its bare essentials, it would look like the following:

```
AnalogInit [
  If 1 AnalogInMain;
  [
     CriticalSection(
```

```
      Root = Self       { This value must be set to self for all
                          points. It is used by the parameter editing
code. };
       { Set up initial values }
      Refresh();
    );
  ]
]
```

Note that the declaration of Root = Self; and the call to Refresh(), must be enclosed in a CriticalSection.

In the simple example, no checking is done to ensure that the expression manager or the alarm manager has started. If you plan to allow expressions for tag parameters (as is commonly done) then you must wait for the Expression Manager. If you plan to include a built-in alarm then you must wait for the Alarm Manager.

The actual initialization state for the Analog Status tag, which includes both of those features, is as follows:

```
AnalogInit [
  If \AlarmManager\Started && \ExpressionManager\Started Ana-
logInMain;
  [
     CriticalSection(
      Root = Self   { This value must be set to self for all
                      points. It is used by the parameter editing
code. };
        { Set up initial values. The refresh sub-module takes care of
I/O, updating the tag's value. }
      Refresh();
    );

    Started = 1;    { Public variable, may be checked by other mod-
ules that depend on this one having started. }
  ]
]
```

A tag's Main state will be more complex, depending on the tag's function. Common tasks include updating the tag's value and other variables. The following partial example is taken from the Analog Status tag.

```
AnalogInMain [
  SitePoint = Scope(Root, DeviceTag);
  Style     = PickValid(Scope(Root, StyleTag), Variable(\#SYSTEM_
STYLE));

  LowScaleValue = PickValid(RangeMin, ScaledMin);
  HighScaleValue = PickValid(RangeMax, ScaledMax);
```

```
]
```

Other tasks include monitoring the tag's link to the I/O device driver, calculating scale, updating the value in an OPC server, etc. In later topics within this chapter, you will see code to develop this state further.

Related functions:

... CriticalSection

ValueSyncService

The role of the value synchronization service (i.e. ValueSyncService) is to allow a tag to register a list of named variables to be kept in synch. The following VTScada tags use this service:

- Totalizer
- Counter
- Selector Switch
- Historian
- History Statistics
- Rate of Change

This service is meaningful only for tags. ValueSyncService is an alternative to the NetworkValues service, which at times can be undesirable for speed, memory or synchronization reasons. For example, NetworkValues do not allow you to control the RPC frequency or when to write to disk but ValueSyncService does.

The service provides startup synchronization of all registered variables. It also provides synchronization on new tag creation while the application is running (online tag creation). In the case of online tags, the values are always synchronized with the primary server for the service. The service supports clients of clients.

Note the following tasks:

- You must explicitly RPC (Remote Procedure Call) the value, unlike NetworkValues.

- You must explicitly persist the value, again unlike NetworkValues.

There are two synchronization situations:

1) Start-up Synchronization:

When a Client comes online or re-syncs with a Server, it will get the Primary server's version of the variables.

2) Online Tag Synchronization:

A new tag is created on a machine, and an EditLockoutManager update is performed. The convention for this is that all machines will sync from the primary server's values, regardless of who actually created the tag first. Thus, whatever happens to be the value of the variable when the tag comes alive on the primary server; that is what is synchronized.

API

\ValueSyncService\Register

Description:	Called by a tag in order to create a list of variables which are to be kept in sync.
Returns:	Nothing (sets PtrRegisterDone when complete)
Usage:	Script Only.
Format:	\ValueSyncService\Register(TagObj, PtrRegisterDone, TagVars)
Parameters:	

TagObj

Required. The object value of the tag.

PtrRegisterDone

Required. A pointer to a variable. Used to return the result of the operation. The value will be set to 1 to indicate that the variables have been synchronized.

TagVars

Required. An array of the variable names to be synchronized.

Example:

As used in the initialization state of the Selector Switch:

```
If \ValueSyncService\Started WaitRegister;
  [
    Root = Self;
    { Register with the ValueSyncService - save processing by registering for the first instance only }
    IfThen(!Valid(SyncedVars),
      { SyncedVars is a SHARED array, we must be the first instance }
      SyncedVars = New(2);
      SyncedVars[0] = "SaveValue";
      SyncedVars[1] = "RequestedValue";
    );
    \ValueSyncService\Register(Root, &RegisterDone, SyncedVars);

    { Set up initial values }
    Refresh();
    {… }
  ]

WaitRegister [
  If RegisterDone Main;
]
```

The Refresh Module

Your tag's Refresh module is called by VTScada when the tag starts and whenever any of the tag's parameters change. It is responsible for ensuring that new values are of the correct type for each parameter. In the case of tags that perform I/O, the Refresh module does not read from equipment, but does send an AddRead() call to the driver. See Linking to a Driver for more information on I/O.

It is extremely important that the Refresh module contain a Return statement. The module will be called as a subroutine and must therefore have a return statement, even if it does not actually return any value.

An example of the code used to update one of a tag's parameters follows. Here, the Analog Status tag's scan rate is being refreshed:

```
ScanRate = PickValid(Cast(ScanRate, 3), GetDefaultValue(&ScanRate));
```

If the parameter is set using a PTypeToggle in the user interface (Constant / Expression / Tag), then it should be evaluated using the ExpressionManager's SafeRefresh method:

```
\ExpressionManager\SafeRefresh(&AlarmLo, Parms[#AlarmLo]);
\ExpressionManager\SafeRefresh(&EnableLogging, Parms[#En-
ableLogging]);
```

The module will always have one parameter, "Parms," which is a pointer to an array of the tag's parameter values prior to being modified by the user. The tag's actual parameters will have already changed by the time "Refresh" is called. The "Parms" array can be used to test for changes in the values, and take appropriate action based upon the changes
A portion of the Analog Status tag's refresh module is shown here as an example. Note that, the portion of a Refresh module that relates to reading or writing data to/from a PLC is covered in the topic, Linking to a Driver.

```
<
{====================== AnalogStatus\Refresh
==========================}
{ This subroutine called at startup and whenever the point's para-
meters }
{   change
}
{=========================================================================-
===}
Refresh
(
  Parms                    { Array for parameters prior to their
change };
)

[
  NeedValueUpdate = FALSE;
]

Refresh [
  If 1;
  [
    ScanRate = PickValid(Cast(ScanRate, 3), GetDefaultValue
(&ScanRate));


{*************************************************************}
    { Scaling
```

```
}

{*****************************************************************}
    UnscaledMin = PickValid(Cast(UnscaledMin, 3 { Float }),
                            GetDefaultValue(&UnscaledMin));
    UnscaledMax = PickValid(Cast(UnscaledMax, 3 { Float }),
                            GetDefaultValue(&UnscaledMax));
    ScaledMin   = PickValid(Cast(ScaledMin,   3 { Float }),
                            GetDefaultValue(&ScaledMin));
    ScaledMax   = PickValid(Cast(ScaledMax,   3 { Float }),
                            GetDefaultValue(&ScaledMax));

    IsText = UnscaledMin == UnscaledMax;


{*****************************************************************}
    { Units
}

{*****************************************************************}
    { Provide default Units (%) for new points
}
    Units = HookPointUnits = Cast(Units, 4 { Text });


{*****************************************************************}
    { ManualValue
}

{*****************************************************************}
    ManualValue = Cast(ManualValue, 3 { Float });
    IfThen(ManualValue != Parm[#ManualValue] ||
           Valid(ManualValue) != Valid(Parm[#ManualValue]),
      NeedValueUpdate = TRUE;
    );

{*****************************************************************}
    { Questionable
}

{*****************************************************************}
    Questionable = PickValid(Cast(Questionable, 0 { Boolean }),
                             GetDefaultValue(&Questionable));


    Return(0);
  ]
]
{ End of AnalogStatus\Refresh }
>
```

Refresh is called prior to the tag's data being written to the tag properties database and is passed an array of the previous parameter values for the tag. The current parameter values for the tag have already been changed by the time the Refresh module has been called. Refresh

provides a way to reduce the memory requirements of the tag by reducing the number of active variable references and code required in the tag's main module, and thereby reduces the amount of memory required for each instance of the tag.

TagShutdown Module

If you need certain tasks to run whenever the tag stops, whatever the reason, then you may add the module, TagShutdown.
When any event causes a tag to stop, VTScada will look for a module named TagShutdown in that tag and execute the code found there. No parameters should be defined. This module must be used as a subroutine.

Examples:

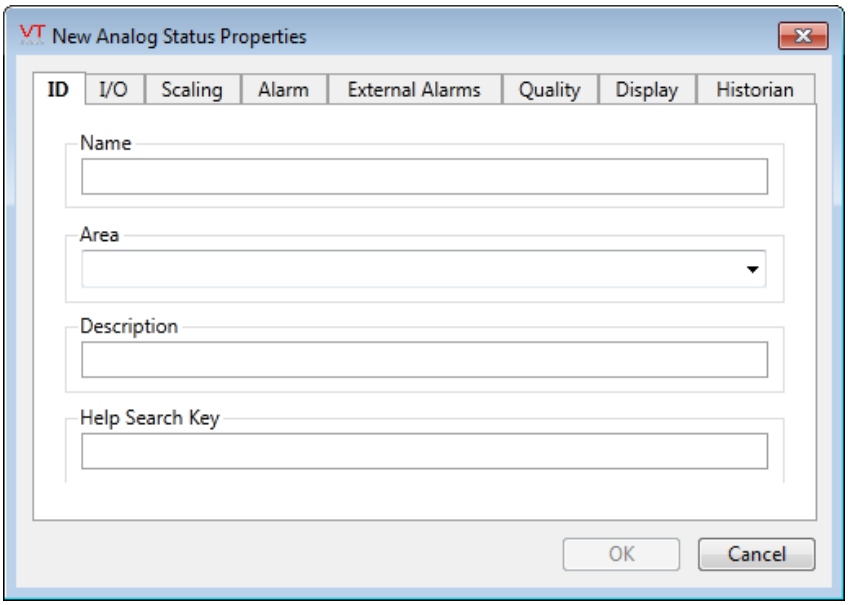
```
{============================ TagShutdown
==============================}
{ Called when the tag is slain - due to either tag stop or app shut-
down.}
{ Makes note of the time when the shutdown occurred. This must be a
}
{ subroutine. }
{=============================================================================-
===}
TagShutdown
Main [
  If 1;
  [ { Save the time of shutdown }
    SavedTime = CurrentTime();
    Return(Invalid);
  ]
]
{ End of TagShutdown }
>
```

Related Information:

... The Refresh Module - Called by VTScada when the tag starts and whenever any of the tag's parameters change

Tag Configuration Folders

Every tag instance will require a user-configured name, area and description at a minimum. I/O connection details and other configuration details will usually be required as well, depending on your tag's purpose.



Configuration is done through a set of configuration folders as shown in the image. VTScada provides the basic folder structure – you need only program the data input fields for each tab and ensure that those are linked to your tag. You will not call the configuration folder directly with your tag code since VTScada handles that task when you click on the Properties button of the Tag Browser.
For most parameter-entry fields and selection-lists, there are helper functions. These are collectively known as the "p-functions".

Related information that you may need:

...Declaring the Configuration Folder Module

...Switching Tabs

...Configuration Tab Contents

...Adding Expression Support for Parameters

...Rules for Config Folders

Declaring the Configuration Folder Module

The code for the configuration folder is typically stored in a TagNameCnf.src file. This is commonly declared by adding a line to the [PLUGINS] class of your AppRoot.src file, assigning a variable name to the source file.
For example, given a tag named "MyTag" stored in the file "MyTag.src" and having a configuration folder in "MyTagCnf.src".
In the AppRoot.src file of the application, add the following to the [PLUGINS] class:

```
  [ (PLUGINS)
    MyTagConfig       Module "MyTagCnf.SRC";
```

In the file MyTag.src, add the following to the Variables declaration section of the module. Note that the assigned name must be "ConfigFolder".

```
[ (PLUGINS)
    Shared  ConfigFolder  =  "MyTagConfig";
```

These two declarations provide VTScada with everything it needs to find your tag's configuration folder. In the next few topics, you will see the code for a sample configuration folder.

The Configuration Folder Module

The configuration folder module is responsible for drawing the contents of the tabbed dialog box used to edit a tag instance's properties. It must perform three tasks:

- Define the tab labels
- Respond to users requests to change from one tab to another
- Display the correct fields in each tab

Additionally, each configuration module must contain the following states:

- An initialization state. Used to initialize variables and pass control to the Switch state.
- A Switch state. Used to switch from one tab's state to another, thereby allowing the user to switch tabs in the panel.

- One state for each tab. These states contain the GUI functions to display the parameter fields.

The Parameters Section

Every configuration module will have the same parameters section:

```
(
  Parms                { Pointer to array of parameters
};
  Current              { Currently selected tab (starts at 0)
};
  PtrWaitClose         { Pointer to FLAG - TRUE when wait to close
};
  OKPressed            { OK Pressed from Properties Dialog
};
  CancelPressed        { Cancel Pressed from the properties Dialog
};
  ParmsData            { Parameter data
};
  OldParms             { Old parameters
};
  OldParmsData         { Old parameter data
};
  ParmsReady           { Pointer to the array of
parameter ready flags };
)
```

- The parameters array is used by VTScada to link your tag parameters to this module.
- VTScada sets the value of Current when a user clicks on a tab in the configuration panel. You will need to watch Current to know when to switch tabs.
- PtrWaitClose is used by the VTScada code that handles the dialog box's Close button. It enables the code to finish all work before the dialog closes.
- OKPressed is similar, but ensures that the values are written to the parameter array.
- CancelPressed is set if the user cancels the configuration session. This allows you the chance to reset any values that may have been changed before cancel was pressed.
- OldParms stores the original values of the parameter fields, prior to the user editing them in the configuration panel.
- ParmsReady is used in types derived from Context tags and tracks if each parameter is ready (TRUE) or in the process of being edited (FALSE).

The Variables Declaration Section

The following are standard declared variables in a configuration folder:

- Width – Normally declared as a constant, VTScada programmers set this value for their convenience when placing objects on the dialog. The height is set automatically by VTScada to accommodate all of the fields in the tallest tab of the dialog.
- Trigger – Used by VTScada. Will be required in the code for each input field.
- EditOK – Should be set according to whether the current user has tag modification privileges, then used in the FocusID field of every input control. (A value of 0 for FocusID disables a p-control.)
- Tab Label Names – The text to display at the top of each tab.

The tab labels are declared in a Class 1 block. The text you provide for each label name will be used as the default if not otherwise specified in the application's Settings.Dynamic file. This system enables easy translation to other languages by changing application properties rather than code.

The order in which the labels are provided will match the order of the tabs from left to right. For example, here is the tab labels declaration from the Analog Status tag:

```
[(1){ class declaration }
  IDTabLabel        = "ID";
  IOTabLabel        = "I/O";
  ScalingTabLabel   = "Scaling";
  AlarmTabLabel     = "Alarms";
  ExtAlmTabLabel    = "External Alarms";
  QualityTabLabel   = "Quality";
  OrderTabLabel     = "Order";
  HistorianTabLabel = "Historian";
]
```

For each tab, you must provide a state containing the graphics statements to be shown in that tab. See, Configuration Tab Contents.

Note: The first tab in every dialog box must be the ID tab.

Configuration Module Initialization

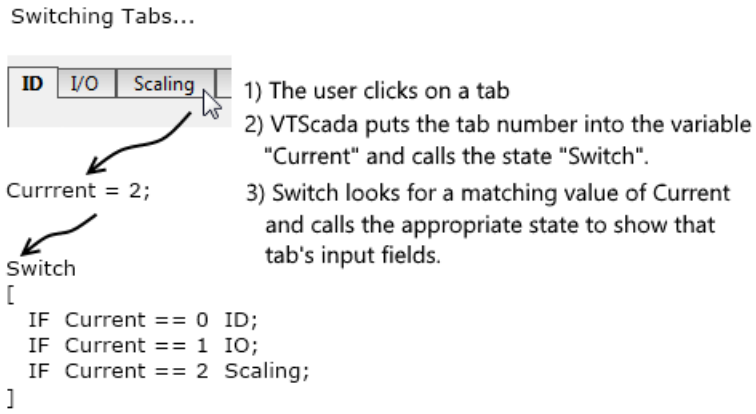
The initial state of the configuration module (commonly named, "Init") will set initial values for variables where required and pass control to the

Switch state. If no initialization is required, the state will always be written as follows:

```
Init [
  If 1 Switch;
  [
    {***** Initialize the wait close flag to not wait *****}
    *PtrWaitClose = 0;
  ]
]
```

Switching Tabs

A switching state must always be part of your configuration module. Commonly named "Switch" this state examines the value of "Current" (set by the user clicking on a tab) and passes control to the appropriate tab's state. The names that you assign to the states need not match the labels, but should be close enough to make sense to anyone reading your code.



For example, here is the Switch state code from the Analog Status tag:

```
Switch [
  If Current == 0 ID;
  If Current == 1 IO;
  If Current == 2 Scaling;
  If Current == 3 Alarms;
  If Current == 4 ExternalAlarms;
  If Current == 5 Quality;
  If Current == 6 Order;
  If Current == 7 Historian;
]
```

Tabs are numbered by VTScada starting at the left with 0.

Configuration Tab Contents

The contents of each tab are provided by a state that will run when the user clicks on the matching tab. Each tab's state code must include a transfer to the Switch state when the value of Current changes.
The state should also update the value of PtrWaitClose based on the Trigger variable. Both are required by the VTScada code that controls the overall dialog box.
The general appearance of a tab state is as follows:

```
TabName [
  If Current != 0 && !*PtrWaitClose Switch;

  *PtrWaitClose = PickValid(Trigger, 1) == 0;

  {***** GUITransforms for each field in the tab. *****}
  GUITransform(…
]
```

A variety of parameter-setting functions have been created to simplify the task of creating the data-entry fields for the parameters. By using these functions in your GUITransforms, you will also standardize the appearance of your tabbed dialog boxes.
As an example, ID tab of nearly every VTScada tag's configuration panel will be similar to the following:

```
ID [
  { User selected a different tab to display }
  If Current != 0 && ! *PtrWaitClose Switch

  { Let the caller know when its ok to close }
  *PtrWaitClose = PickValid(Trigger, 1) == 0

  { Help topic to display when user presses F1 }
  SetHelp(Self(), \DevHelpFile, 00000 { HelpID 00000 })

  { Name of the tag }
  GUITransform(30, 90, Width - 30, 45 { Boundaries of transform },
               1, 1, 1, 1, 1          { No scaling },
               0, 0, 1, 0             { No movement; visible;
reserved },
               0, 0, 0                { Not selectable },
               \DialogLibrary\PEditName(Trigger));

  { Tag area }
  GUITransform(30, 300, Width - 30, 100 { Boundaries of transform
},
               1, 1, 1, 1, 1            { No scaling
```

```
},
                0, 0, 1, 0                { No movement; visible;
reserved },
                0, 0, 0                   { Not selectable
},
                \DialogLibrary\PAreaSelect(1 { can edit }, 2 { ID },
Parms[1] { Init },
                                            1 { Bevel }, 0 { VertAlign
},
                                            1 { AlignTitle }, 1 {
ParmNum },
                                            Trigger { Trigger },
OKPressed))

  { Tag Description  }
  GUITransform(30, 200, Width - 30, 155 { Boundaries of transform
},
                1, 1, 1, 1, 1             { No scaling
},
                0, 0, 1, 0                { No movement; visible;
reserved },
                0, 0, 0                   { Not selectable
},
                \DialogLibrary\PEditField(2, \DescriptionLabel, 4 {
text }, 3 { ID }, Trigger { trigger }));

  { Help key }
  GUITransform(30, 255, Width - 30, 210 { Boundaries of transform
},
                1, 1, 1, 1, 1             { No scaling
},
                0, 0, 1, 0                { No movement; visible;
reserved },
                0, 0, 0                   { Not selectable
},
                \DialogLibrary\PEditField(\#HelpKey, \HelpSearchKeyLa-
bel, 4 { text }, 4 { ID }, Trigger { trigger }))
]
```

Related functions:

For further information about the various parameter-setting functions, see:

PAddressEntry	PAlmPriority	PAreaSelect	PCheckBox
PColorSelect	PColorEdit	PContributor	PDroplist
PEditfield	PEditName	PHSliderBar	PHueSelect
PIPAddressList	PMultiCheckBox	POverride	PPageSelect
PRadioButtons	PSecBit	PSelectObject	PSpinbox
PTypeToggle			

Alarm Tab Notes

If you intend to allow developers to set the comparison function used when triggering the alarm (>, =, <=, etc) then you should create a dictionary of the function codes. Declare it as a shared variable in the tag module:

```
SHARED FunctionCodes { Function code strings };
```

Then populate the dictionary in the Init state, again in the tag module:

```
MyTagInit [

  If \AlarmManager\Started && \ExpressionManager\Started MyTagMain;
  [
    { If (and only if ! ) you are allowing the developer to choose
the comparison function, create list of function codes }
    IfThen(!Valid(FunctionCodes),
      FunctionCodes = Dictionary();
      FunctionCodes["<"]    = \AlarmManager\ALM_FUNC_LESS_THAN;
      FunctionCodes["<="]   = \AlarmManager\ALM_FUNC_LESS_EQUAL;
      FunctionCodes[">="]   = \AlarmManager\ALM_FUNC_GREATER_EQUAL;
      FunctionCodes[">"]    = \AlarmManager\ALM_FUNC_GREATER_THAN;
      FunctionCodes["="]    = \AlarmManager\ALM_FUNC_EQUAL;
      FunctionCodes["=="]   = \AlarmManager\ALM_FUNC_EQUAL;
      FunctionCodes["!="]   = \AlarmManager\ALM_FUNC_NOT_EQUAL;
      FunctionCodes["<>"]   = \AlarmManager\ALM_FUNC_NOT_EQUAL;
      FunctionCodes["&"]    = \AlarmManager\ALM_FUNC_AND_WITH;
      FunctionCodes["&&"]   = \AlarmManager\ALM_FUNC_AND_WITH;
      FunctionCodes["|"]    = \AlarmManager\ALM_FUNC_OR_WITH;
      FunctionCodes["||"]   = \AlarmManager\ALM_FUNC_OR_WITH;
      FunctionCodes["^"]    = \AlarmManager\ALM_FUNC_XOR_WITH;
      FunctionCodes["NAND"] = \AlarmManager\ALM_FUNC_NOT_AND_WITH;
      FunctionCodes["NOR"]  = \AlarmManager\ALM_FUNC_NOT_OR_WITH;
    );

    CriticalSection(
      Root = Self();
      Refresh();
    );
    Started = 1;
  ]
]
```

In the module for the configuration panel, create the following variables:

```
FindFunction Module "FindFunc.WEB" { Finds function in FuncList };
FuncList { List of valid functions };
FuncIndex { Index into the FuncList array };
FuncValues { Values of the function descriptions };
FuncType { The long text string version of Function };
```

Then in the panel's Init state, initialize the arrays:

```
Init [
  If 1 Switch;
  [
  {***** Set up the function list *****}
    FuncList = New(12);
    FuncList[0] = \NoFunctionLabel;
    FuncList[1] = Concat(\LessThanLabel, " <");
    FuncList[2] = Concat(\LessThanEqualLabel, " <=");
    FuncList[3] = Concat(\GreaterThanLabel, " >");
    FuncList[4] = Concat(\GreaterThanEqualLabel, " >=");
    FuncList[5] = Concat(\EqualToLabel, " = ", \OrLabel, " ==");
    FuncList[6] = Concat(\NotEqualToLabel, " != ", \OrLabel, " <>");
    FuncList[7] = Concat(\ANDedWithLabel, " & ", \OrLabel, " &&");
    FuncList[8] = Concat(\ORedWithLabel, " | ", \OrLabel, " ||");
    FuncList[9] = Concat(\XORedWithLabel, " ^");
    FuncList[10] = Concat(\NotANDedWithLabel, " ! ( && )");
    FuncList[11] = Concat(\NotORedWithLabel, " ! ( || )");
    FuncValues = New(12);
    FuncValues[0] = "";
    FuncValues[1] = "<";
    FuncValues[2] = "<=";
    FuncValues[3] = ">";
    FuncValues[4] = ">=";
    FuncValues[5] = "=";
    FuncValues[6] = "!=";
    FuncValues[7] = "&";
    FuncValues[8] = "|";
    FuncValues[9] = "^";
    FuncValues[10] = "NAnd";
    FuncValues[11] = "NOr";
    {***** Find the current function type *****}
    FuncType = FindFunction(Parms[\#Function]);
    FuncIndex = LookUp(FuncList[0], 12, FuncType);
    { ... }
```

Finally, in the state for the tab where the alarm function is to be selected, display the list:

```
{***** Function *****}
GUITransform(30, 155, 470, 110,
             1, 1, 1, 1, 1,
             0, 0, 1, 0,
             0, 0, 0,
             \DialogLibrary\PDropList(\#Function, \FunctionLabel,
                                      FuncList, 0, FuncIndex {
Labels, CanEdit, Index },
                                      HasPriv ? 11 : 0, 0 { FocusID,
No Trigger },
                                      FuncIndex, 1, 0, 1 { Init,
DrawBevel, VertAlign, AlignTitle },
                                      FuncValues { Return Values }));
```

Related Information:

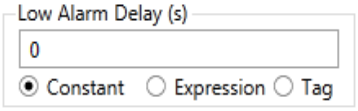
...Adding Alarms to Custom Tags

...Alarm Manager Function Constants

Adding Expression Support for Parameters

Support for the Expression Manager is what makes the difference between this configuration field:

And the following:



If your tag includes support for expressions, you must ensure that the ExpressionManager is started in the tag's initialization state before any other actions take place. This is commonly done as follows (example taken from the Analog Status tag).

```
AnalogInit [
  If \AlarmManager\Started && \ExpressionManager\Started Ana-
logInMain;
  [

    CriticalSection(
      Root = Self       { This value must be set to self() for all
                          points. It is used by the parameter editing
                          code.                                    };
       { Set up initial values }
      Refresh();
    );
  ]
]
```

This example shows a check for the Alarm Manager being started as well as the Expression Manager. The process of adding custom alarms to your tag is discussed later in this chapter.

In the tag's Refresh module, the \ExpressionManager\SafeRefresh function is used to handle changes to any parameters that may use a tag or expression for their value.

```
\ExpressionManager\SafeRefresh(&AlarmLoDelay,   Parm[#AlarmLoDelay]);
```

(see: Expressions as Tag Parameters for more details.)

Finally, in the configuration folder, the PTypeToggle statement is used to display the input field with radio selection buttons.

```
{***** Low Delay *****}
GUITransform(30, 294, WIDTH/2 - 5, 236,
             1, 1, 1, 1, 1            { No scaling
},
             0, 0, 1, 0               { No movement; visible;
reserved },
             0, 0, 0                  { Not selectable
},
             \DialogLibrary\PTypeToggle(\#AlarmLoDelay, "Numeric"
                                            { point type },
                                        \LowAlarmDelayLabel,
                                        EnableLo ? 15 : 0 { to 17 -
ID },
                                        0 { top align }, 1 { align
title },
                                        0, Invalid { limits }, Trig-
ger,
                                        1 { Allow Expression }));
```

Rules for Config Folders

- A module must be defined to handle the drawing of the contents of the tabbed tag properties folder that enables users to configure the properties of tags belonging to this tag type while the application is running.
- The ConfigFolder module is configured with one state per tab on the tag properties dialog. The tabs contain sets of the tag type's parameters organized into logically consistent groups. When the user clicks on a tab, the value of the "Current" parameter (see above) changes to reflect the index of the selected tab, and the ConfigFolder module changes states to draw new data entry fields on the active tab of the tag properties folder
- A "Switch" state must be provided to transfer control to the appropriate state when the value of Current changes.
- The data entry fields on each tab of the tag properties folder are drawn primarily with a set of tools that are contained in a module named, "DialogLibrary". These tools enable you to draw text and numeric entry fields, drop-down lists, and radio buttons.
- When defining the labels to appear on the tabs of the tag properties folder for your custom tag type, first define a list of class 1 variables that refer to a set of tab name variables to be found in the application's configuration. The order of these variables should be the order in which the tabs are displayed

in the tag properties folder. Each class 1 variable should be provided with a default value; therefore, if there is no corresponding value set in the application's configuration, this default value can be used for the tab label. Note: The class 1 variables do not directly represent the text label to be displayed on each tab; rather, the configuration variables they reference contain the text labels for the tabs. This organization enables you to later translate the labels into another language, or otherwise modify their text without having to change the application's code.

Create or Assign Tag Widgets

VTScada provides an extensive selection of widgets that you may use with your tags. You can also write a custom widget for a tag if none of the built-in methods meet your needs.

To use the tag widgets provided by VTScada, you need only declare the ones you wish to use in the [(GRAPHICS)] class of your tag module's variables declaration section. A complete list of the available widgets can be found in the chapter, Drawing Tags.

The following example is taken from the Analog Status tag: (list reduced here to save space)

```
  [ (GRAPHICS)
    Draw        Module     { Standard Draw Module for this point
};
    Shared TopBar;
    Shared RightBar;
    Shared LeftBar;
    Shared BottomBar;
    Shared Number;
    Shared DrawText;
    Shared Meter1;
    Shared Meter2;
    Shared Meter3;
    Shared Meter4;
    Shared Compass1;
    Shared AnimatedBitmap;
    Shared TwoColorBar;
    Shared ColorFill;
  ]
```

The top line in the list is a call to a local module that supplies a custom widget (named Draw) for this tag.

Note for Style-Tag Aware Widgets

The Status Color Indicator widgets and Indicator Light widgets are typically used to display a tag's value using the colors defined in the Digitals tab of the associated Style Settings tag. But, for tags in the ports group, drivers group, the Modem tag and the SMS Appliance tag, these widgets take the tag's value to represent an error state, and use the colors defined in the Errors tab of the style tag.

To inform the widget that your tag's value should be interpreted as an error indicator rather than a status value, you should add the following three parameters to the tag.

- *ValueIsErrorStatus* as a Boolean. Set TRUE to treat the tag's value as an error status condition and to use the No Error and Error colors of the Style Settings tag.
- *ValueIsErrorAbove* as a numeric. This is the value above which the tag will be treated as being in error. Defaults to 0.
- *ValueIsErrorBelow* as a numeric. This is the value below which the tag will be treated as being in error. Defaults to 0.

Related information that you may need:

...Create a Custom Tag Widget

...Widget Parameters

...Example – Parameters for the Analog Status's Draw Widget

...Edit Mode versus Run Mode

...The Properties Panel

...Widget States

...Indicating Questionable and Manual Data

...Rules for Tag Widgets

Create a Custom Tag Widget

It is often the case that a custom tag will have one or more custom widgets. You can build these using the information that follows.

To begin, declare the drawing module in the [(GRAPHICS) section of the tag, as described in the preceding topic. (see: Create or Assign Tag Widgets)
Each widget must have its own module, where there is one widget per module. The name of the module will be taken as the name of the widget unless you specify otherwise with a constant declaration named "DrawLabel".

```
…
  Constant  DrawLabel = "MyDrawMethodName";
…
```

The following are common elements of a custom widget. Required elements are noted.

- Optional parameters for configuring the object.
- An optional initialization state that sets default values if required (colors, fonts, sizes, etc.) and call's the module's main state.
- The main state should include the steady state graphics command that define the drawing object.
- The main state must also launch a call to the tag's Common module.
- A submodule named "Panel" should be included if the widget has parameters. This is used by the VTScada Graphics Editor when the drawing object is in editing mode. If a Panel method is not supplied, VTScada will generate a basic configuration box for the parameters when the object is placed on the screen.
- Panel must be declared with one parameter, Parms. This will be a pointer to an array that contains the initial values of the tag's parameters and that will be modified to contain the values entered by the user while editing.

Each of these elements will be described in the following topics.

Note: You will need to create a MenuItem tag for your widget if you want to see it in any of the Idea Studio palettes.

Related information that you may need:

...Widget Parameters

...Edit Mode versus Run Mode

...The Properties Panel

...Widget States
Refer to the VTScada Developer's Guide for:

...Right-Click - Editing the Palette MenuItem Tags - Create an entry for your custom widget

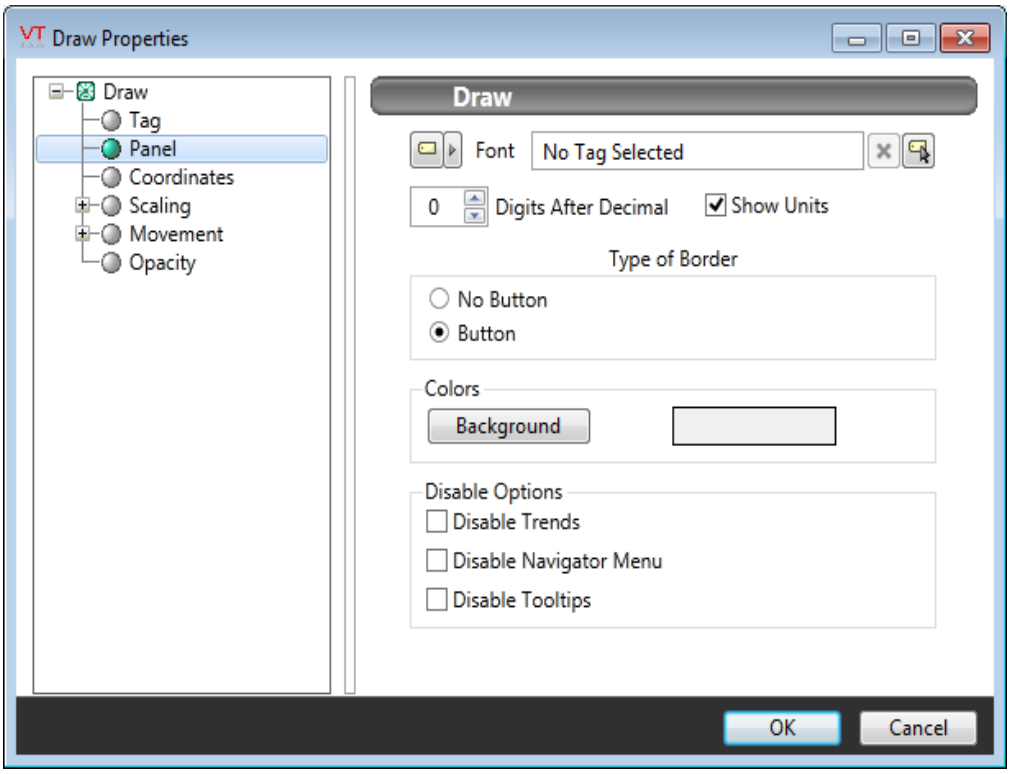
...Menuitem Tag - Menu Item details and reference

Widget Parameters

The parameters for a drawing object are structured much like those of a tag. They must be declared in a parameters section and they must be assigned constants in the variables declaration section. The main difference is that widget parameters are not stored in a database and need not be assigned
Three common parameters that you may want to include are DisableTrend, DisableNavigation and DisableTooltip. As their names imply, these are used to disable user interface features of the Common module when required.

Example – Parameters for the Analog Status's Draw Widget

As an example, the Analog Status tag has the following options in its Draw widget:



The matching parameters are declared as follows. (Parameters and variables for features that have not yet been covered in this chapter are excluded).

Note the presence of the word "Panel" in the menu. The contents of this display are controlled by a submodule of the widget named "Panel". You will need to create this module for your tag.

The four numeric values after the module name and before the parameter list are used to specify the size of the module reference box. When VTScada calls a module within a GUITransform, it scopes in and gets the bounds of largest of the GUI within that module. Setting the default graphic size avoids problems when there may be non-GUI graphics or you want the drawing area bigger to be than the GUI's within it. See: Module Reference Boxes.

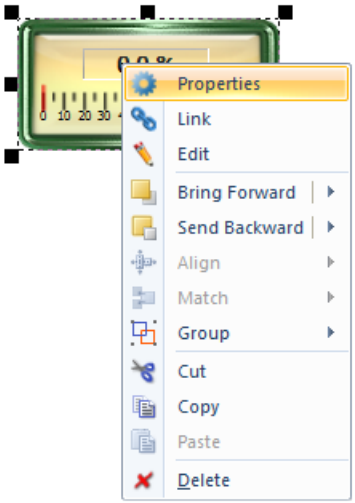
```
<
{================================ Draw
==================================}
{ Graphic module to display Text Description and Value of the Input
}
{=================================================================-
===}
Draw
```

```
(0, 30, 200, 0)
(
  Digits                    { Digits after the decimal point
};
  ShowUnits                 { TRUE if display the units
};
  FontVal                   { Font to display text in
};
  Border                    { Border options: 0 = no button, 1 = but-
ton };
  BgndColor                 { Color for background inside the button
};
  DisableTrend              { Flag to disable trend windows
};
  DisableNavigation         { Flag to disable navigator menu
};
  DisableTooltip            { Flag to disable tool tips
};
)

[
  Panel Module               { Parameter editing module
};
  Offset                     { Offset for graphics, based on Width
};
  Color                      { Background color based on status of
point };
  TextColor                  { Text color based on questionable data
};
  Buff                       { Text buffer holding string to display
};
  ChosenFont                 { The font value chosen by the user
};
  DrawFontObj                { Object value of font to draw with
};
  Format                     { Format for the number
};

  { Parameter definitions }
  Constant #Digits            = 0;
  Constant #ShowUnits         = 1;
  Constant #FontVal           = 2;
  Constant #Border            = 3;
  Constant #BgndColor         = 4;
  Constant #DisableTrend      = 5;
  Constant #DisableNavigation = 6;
  Constant #DisableTooltip    = 7;

   { Panel dimensions }
  Constant #PanelWd = 320;
  Constant #PanelHt = 6 * Space + TitleSpace + 3 * CheckHt + 242;
]
```

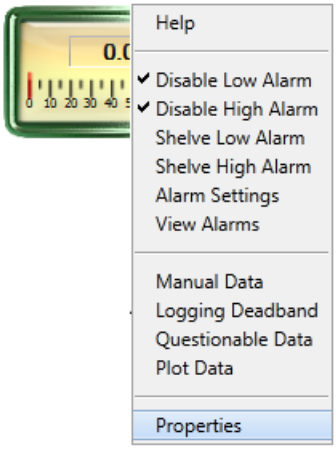
Edit Mode versus Run Mode

When the application is in edit mode (Idea Studio), widgets should respond to user clicks differently than when the application is in run mode.

Right-click while in Edit mode



Right-click while in Run mode



You can determine which mode the application is in by querying, ParentWindow()\Editing. VTScada will return a value of TRUE or FALSE depending on the mode. You can then code your method to examine the \Editing value before responding to a user's action.

An example can be seen with Page Hotboxes. When in Edit mode, these display as a yellow rectangle, but when in run mode, they are invisible until the user hovers the mouse pointer over them. You can emulate this behavior by adding a GUIRectangle to your widget and coding its Visibility parameter to examine the current value of ParentWindow()\Editing.

The Properties Panel

If the widget includes parameters, it is strongly recommended that you supply a Panel module. The purpose of this module is to control the layout of the parameter elements in the properties dialog when the widget is being edited.

The Panel module takes two parameters: Parms, which is a pointer to an array containing the widgets parameters and PtrWaitClose, which is used by VTScada when the properties dialog is being closed.

The Panel module will normally also have one internal variable, Trigger. This will be used by VTScada when each field in the panel is edited by the user.
Like the tag configuration dialog, the P* functions are commonly used for the data entry fields.
For example, the entirety of the Analog Status tag's widget panel, as shown in the preceding topic, is as follows:

```
<
{============================== Panel
==============================}
Panel
(0, #PanelHt, #PanelWd, 0)
(
  Parms;
  PtrWaitClose { Pointer to Flag - TRUE to wait to close dlg };
  ParmDefs { Array of page parameter definition structures };
  DialogRoot { The calling dialog window };
)
[
  Trigger { Edit fields trigger };
  SubWaitClose { WaitClose flag for the datasource editor };
  Height { Height of panel };
  PageInst { Page instance };
  ContPageInfo { Container page information };
  Modules { Array of modules to load in the ParameterEdit };
  Parameters { Array of parameters for ParameterEdit modules };
  ParmEditObj { Object value of the ParameterEdit control };
  PEBottom { Bottom of the ParameterEdit };
  FontParm; OldFontParm { Used for ParameterEdit };
]
Init [
  If 1 PanelMain;
  [
    {***** Set defaults *****}
    Parms[#Digits] = PickValid(Parms[#Digits], 1);
    Parms[#ShowUnits] = PickValid(Parms[#ShowUnits], 1);
    Parms[#Border] = PickValid(Parms[#Border], 0);
    Parms[#BgndColor] = PickValid(Parms[#BgndColor], \DialogBGColor);
    Parms[#DisableTrend] = PickValid(Parms[#DisableTrend], 0);
    Parms[#DisableNavigation] = PickValid(Parms[#DisableNavigation],
0);
    Parms[#DisableTooltip] = PickValid(Parms[#DisableTooltip], 0);
    { Initialize the wait close flag to not wait }
    *PtrWaitClose = FALSE;
    { Get info about the parms of the container page }
    PageInst = \GetUserSession()\GraphicEditor\GetSelectedTabItem();
    ContPageInfo = GetParamInfo(PageInst);
    { Set up variables for ParameterEdit }
    FontParm =
      OldFontParm = Parms[#FontParm];
    MetaData(FontParm, "Revision") = 0;
    { "Font" ParameterEdit }
```

```
    Modules = New(3);
    Parameters = New(3);
    { Tag option }
    Modules[0] = "ParmEditTag";
    Parameters[0] = New(1);
    Parameters[0][0] = "FontValue";
    { Parameter option }
    Modules[1] = "ParmEditParmValue";
    Parameters[1] = New(6);
    Parameters[1][0] = ContPageInfo;
    Parameters[1][1] = \#VTypeObject;
    Parameters[1][2] = Invalid;
    Parameters[1][3] = New(1);
    Parameters[1][3][0] = "FontValue";
    Parameters[1][4] = 0; { No Scaling }
    Parameters[1][5] = Invalid; { No default value }
    { Expression option }
    Modules[2] = "ParmEditExprNoNormalize";
    Parameters[2] = New(2);
    Parameters[2][0] = 1;
    Parameters[2][1] = ContPageInfo;
  ]
]
PanelMain [
  {***** Let the caller know when its ok to close *****}
  *PtrWaitClose = PickValid(Trigger, 1) == 0 || PickValid
(SubWaitClose, 0) == 1;
  Height = 10*Space + 2*TitleSpace + PEBottom + SpinHt + BtnHt +
3*CheckHt + 65;
  If Watch(0, Height);
  [
    SetPanelRefBox(Self, 0, Height, #PanelWd, 0);
  ]
  {***** Font *****}
  GUITransform(0, 1, 1, 0,
    1 - 0,
    2 * (EditHt + Space),
    #PanelWd,
    1 - 0,
    1, 0, 0, 1, 0, 0, 0, 0,
    ParmEditObj = \ParameterEdit(FontParm { Parameter Value },
    ParmDefs[#FontParm] { Parameter Definition },
    1 { Enable Flag },
    \FontLabel { Title },
    Modules { Array of Parm Edit Modules },
    \Code { Contexts for Edit Modules },
    Parameters { Parameters for Edit Modules },
    TextAttribs(\FontLabel, _DialogFont, 0) { Title Width },
    Invalid { Index value },
    SubWaitClose { Wait to close },
    DialogRoot { Calling dialog window }));
    If FontParm != OldFontParm ||
      Valid(FontParm) != Valid(OldFontParm);
    [
      OldFontParm =
        Parms[#FontParm] = RootValue(FontParm);
    ]
```

```
    PEBottom = PickValid(ParmEditObj\Height, 2 * (EditHt + Space));
    {***** Digits *****}
    GUITransform(0, 1, 1, 0,
      1 - 0,
      Space + PEBottom + SpinHt,
      170,
      1 - (Space + PEBottom),
      1, 0, 0, 1, 0, 0, 0, 0,
      \DialogLibrary\PSpinBox(#Digits, \DigitsAfterDecLabel, 1 { box
on left },
      0, 9 { limits }, 3 { left, centered }, 0 { autosize },
      0 { no edit }, 3 { ID }, Invalid, Invalid, Trigger {Trigger}));
    {***** Show units *****}
    GUITransform(0, 1, 1, 0,
      1 - 180,
      Space + PEBottom + CheckHt + 4,
      #PanelWd,
      1 - (Space + PEBottom + 4),
      1, 0, 0, 1, 0, 0, 0, 0,
      \DialogLibrary\PCheckBox(#ShowUnits, \ShowUnitsLabel, 1 { box
on left },
      3 { left, centered }, 4 { ID }));
      {***** Type of border *****}
    GUITransform(0, 1, 1, 0,
      1 - 0,
      2*Space + PEBottom + SpinHt + 65,
      #PanelWd,
      1 - (2*Space + PEBottom + SpinHt),
      1, 0, 0, 1, 0, 0, 0, 0,
      \DialogLibrary\PRadioButtons(#Border, 8 { to 10 - ID },
      1 { border }, \TypeOfBorderLabel, 1 { btns on left }, 1 { align
title },
      \NoButtonLabel, \ButtonLabel));
    {***** Colors *****}
    \System\Bevel(0,
      5*Space + TitleSpace + PEBottom + SpinHt + BtnHt + 65,
      #PanelWd,
      3*Space + PEBottom + SpinHt + 65,
      \ColorsLabel);
    {***** Background color *****}
    GUITransform(0, 1, 1, 0,
      1 - Space,
      4*Space + TitleSpace + PEBottom + SpinHt + BtnHt + 65,
      260,
      1 - (4*Space + TitleSpace + PEBottom + SpinHt + 65),
      1, 0, 0, 1, 0, 0, 0, 0,
      \DialogLibrary\PColorSelect(#BgndColor, \BackgroundLabel, 1 {
btn on left },
      1 { standard size }, 1 { centered }, 12 { ID }));
    {***** Disable Options *****}
    \System\Bevel(0,
      10*Space + 2*TitleSpace + PEBottom + SpinHt + BtnHt + 3*CheckHt
+ 65,
      #PanelWd,
      6*Space + TitleSpace + PEBottom + SpinHt + BtnHt + 65,
      \DisableOptionsLabel);
    {***** Disable Trends Check Box *****}
```

```
    GUITransform(0, 1, 1, 0,
      1 - Space,
      7*Space + 2*TitleSpace + PEBottom + SpinHt + BtnHt + CheckHt +
65,
      #PanelWd - Space,
      1 - (7*Space + 2*TitleSpace + PEBottom + SpinHt + BtnHt + 65),
      1, 0, 0, 1, 0, 0, 0, 0,
    \DialogLibrary\PCheckBox(#DisableTrend, \DisableTrendLabel,
      1 { box on left },
      3 { left, centered },
      13 { ID }));
    {***** Disable Navigation Check Box *****}
    GUITransform(0, 1, 1, 0,
      1 - Space,
      8*Space + 2*TitleSpace + PEBottom + SpinHt + BtnHt + 2*CheckHt
+ 65,
      #PanelWd - Space,
      1 - (8*Space + 2*TitleSpace + PEBottom + SpinHt + BtnHt +
CheckHt + 65),
      1, 0, 0, 1, 0, 0, 0, 0,
      \DialogLibrary\PCheckBox(#DisableNavigation, \Dis-
ableNavigationLabel,
      1 { box on left },
      3 { left, centered },
      14 { ID }));
    {***** Disable Tooltip Check Box *****}
    GUITransform(0, 1, 1, 0,
      1 - Space,
      9*Space + 2*TitleSpace + PEBottom + SpinHt + BtnHt + 3*CheckHt
+ 65,
      #PanelWd - Space,
      1 - (9*Space + 2*TitleSpace + PEBottom + SpinHt + BtnHt +
2*CheckHt + 65),
      1, 0, 0, 1, 0, 0, 0, 0,
      \DialogLibrary\PCheckBox(#DisableTooltip, \DisableTooltipLabel,
      1 { box on left },
      3 { left, centered },
      15 { ID }));
]
{ End of Draw\Panel }
```

Widget States

Most widget modules will contain two states: one used to initialize the parameter values and one to display the tag's value and other graphics on a page.

The display functions must monitor properties of the tag. If the tag's units or scaling change, the widget will need to update its appearance to match the new format. It should also be able to display the Questionable or Manual Value markers if those exist and are set in the tag. (An

example of the code to do this can be seen in Create a Custom Tag Widget.)

Finally, the tag's main state must launch a call to the tag's Common module if it is to support tool tips, trend windows or right-click navigation menus.

An abbreviated version of the states found in the Analog Status tag's Draw widget is presented here as an example. Features such as the alarm display have not yet been covered in this chapter and so are excluded from the example.

```
Init [
  If 1 Main;
  [
    {***** Set defaults for the parms *****}
    Digits            = PickValid(Digits,    0);
    ShowUnits         = PickValid(ShowUnits, 1);
    Border            = PickValid(Border,    1);
    BgndColor         = PickValid(BgndColor, \DialogBGColor);
    DisableTrend      = PickValid(DisableTrend,      0);
    DisableNavigation = PickValid(DisableNavigation, 0);
    DisableTooltip    = PickValid(DisableTooltip,    0);
  ]
]

Main [

  {***** Work format for display of value *****}
  Format = Concat(Valid(\Value) ? Concat("%0.", PickValid(Digits, 1),
"f")
                                : Concat("*.", MakeBuff(PickValid
(Digits,
                                                        1), 0x2A { * character
})),
                  " ,"
                  PickValid(\Questionable, 0) ? "?" : "",
                  Valid(\ManualValue) ? "!" : "",
                  PickValid(\Questionable, 0) ||
                            Valid(\ManualValue) ? " " : "",
                  PickValid(ShowUnits, 0) ? "%s" : "",
                 );

  {***** Figure out what the text looks like *****}
  If Watch(1, Format, \ScaledMax, \Units);
  [
    {***** Get the full string - number, decimal, units and all
*****}
    Buff       = BuffStream("");
    SWrite(Buff, Format,
           PickValid(PickValid(\ScaledMax, \Value), \Units), \Units);
  ]
```

```
  {***** Convert font point text name into an object and a font
*****}
  If Watch(1, FontVal);
  [
    DrawFontObj = ValueType(FontVal) == 7 { Tag object } ?
                  FontVal : Scope(\Code, FontVal);
  ]
  ChosenFont = Font(DrawFontObj\FontName, DrawFontObj\CharacterSet,
                    DrawFontObj\Height,
                    DrawFontObj\Rotation, DrawFontObj\Weight,
                    DrawFontObj\Italic,   DrawFontObj\Fixed);


  QualityIssue   = PickValid(Scope(Root, Quality)\Value != 0, 0);

  TextColor = PickValid(Questionable, 0) ? \ButtonShadow :
                (Color == \ButtonTextColor ? 15 : \ButtonTextColor);

    { Background Button }
  GUIButton(0, 30, 200, 0,
            1, 1, 1, 1, 1,
            0, 0, Border, 0, 0, 0, 0,
            BgndColor, \ButtonHighlight , \ButtonShadow, -1,
            4, 0, "", "", 0, 0, 1, 2);
    { Description }
  GUIText(4, 30, 200, 0,
          1, 1, 1, 1, 1,
          0, 0, Border, 1,
          0, 0, 0,
          -1, \ButtonTextColor, PickValid(DrawFontObj\Value, \_Dia-
logFont),
          2, 4,   { Alignment }
          \Description);
    { The indented box }
  GUIButton(130, 26, 196, 4,
            1, 1, 1, 1, 1,
            0, 0, Border, 0, 0, 0, 0,
            -1, \ButtonShadow, \ButtonHighlight, -1,
            4, 0, "", "", 0, 0, 1, 2);

    {***** Display the number *****}
  GUIText(0, 1, 1, 0,
          1 - (Border ? 131 : 0), Border ? 25 : 30,
          Border ? 195 : 200, 1 - (Border ? 5 : 0), 1,
          0, 0, 1, 1,
          0, 0, 0,
          Color, TextColor,
          PickValid(DrawFontObj\Value, \_DialogFont),
          4, 4, { Alignment }
          Format, PickValid(\Value, \Units), \Units);

  {***** Display common features *****}
  \Common(0, 30, 200, 0, DisableTrend, DisableNavigation, Dis-
ableTooltip);
]
```

Indicating Questionable and Manual Data

The unlinked widget, question mark and exclamation mark graphics, used to indicate any of these tag states in a widget, are created by the module TagIconMarker.
This module places a set of icons on the screen centered over a given rectangular region. The icon displayed is cycled with each passage of the period, measured in seconds. The images are provided by the module, but you are free to add your own symbols.
Centering the symbol within the user-defined drawing area requires some calculation. For more information about this function and an example taken from a VTScada tag widget, see IconMarker in the VTScada Function List.
The following example places a set of icons on the screen centered over a given rectangular region. The icon displayed is cycled with each passage of the period, measured in seconds.

```
{***** Signify when data is questionable *****}
GUITransform(0, 100, 100, 0,
             1, 1, 1, 1, 1 { Scaling },
             0, 0 { Movement },
             1, 0 { Visibility, Reserved },
             0, 0, 0 { Selectability },
             Variable("Code\Library")\TagIconMarker(\Root, FALSE));
```

Related functions:

...TagIconMarker

... IconMarker

Rules for Tag Widgets

- Drawing modules (modules within the tag that provide a method for displaying the value of the tag on the system pages of your application) must always be declared in the GRAPHICS class.
- Create a Panel module within any of the tag widget modules that have parameters. The Panel module performs a task similar to that of the ConfigFolder module (see number 8 above), except that it is responsible for the editing of the tag widget module's parameters, rather than the parameters for the tag as a whole. A Panel module is not required if the tag's widget module has no

parameters. If a tag widget module doesn't have a Panel module, a default parameter configuration dialog box is instantiated when the object is placed on the screen.

- The Panel module takes two parameters: Parms – A pointer to an array that contains the initial values of the tag's parameters, and that will be modified to contain the values entered by the user while using the tag properties dialog. PtrWaitClose – A pointer to a flag – TRUE to wait to close dialog.
- The Parms parameter is handled in the same manner as the first parameter for the ConfigFolder module (please refer to the Dialog Library Tools section for details on how to build this module to create the dialog box). This dialog box differs from the ConfigFolder dialog in that it is not a tabbed dialog box, but it does have a preview window at the bottom to allow users to view an image of how the completed tag widget will appear when placed on the system page. The size of the dialog box is automatically determined by calling code in VTScada so that it is the minimum sized window that will contain all of the GUI objects used.
- Use the "ParentWindow()\Editing" variable to prevent control actions within the Idea Studio window. The \Editing variable is set to true within the context of the editor window. Preventing control dialogs or other mouse actions from being acted upon when this value is true will prevent unintended control when an object is selected for editing.
- If a widget for an object is normally invisible, you should use the ParentWindow()\Editing variable in the visibility parameter of a GUIRectangle in order to display a yellow box around the perimeter of the area so that it can be seen during editing and can be selected.
- Inside the drawing module, you may define variables called DrawWidth and DrawHeight that can be used to set the width and height of the object placed on the screen during configuration. This is useful in setting the size of an image that is different than the size given by the GUIBitmap reference box. These variables are typically set in the "Main" module, according to the current parameters set for the graphic object.
- Add a constant definition for "DrawLabel" with a default value to each drawing module. This variable's default value specifies the name of a variable in the configuration that defines the label to place on the drawing object's pre-

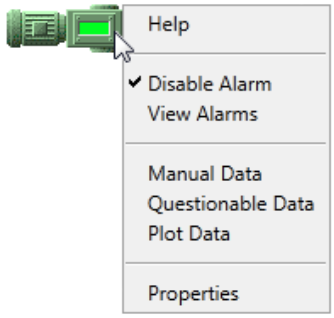
views during configuration. If this variable is not present, the drawing module's name is used.

Common Module

Various user-interface features are common to the VTScada tag widgets. The widgets will issue calls to the Common module of the associated tag in order to provide the contents and functionality of the interface feature. These may include a call to the Navigator module, the ToolTip module, and the PkTrend module. Any external graphic modules must make a call to the Common module within the tag. The Common module takes four parameters that define the area of the screen occupied by the drawing object and three that allow an instance of a tag to disable the user interface elements. The seven parameters of the Common module are:

- Left – Any numeric expression for the object's left side coordinate.
- Bottom – Any numeric expression for the object's bottom coordinate.
- Right – Any numeric expression for the object's right side coordinate.
- Top – Any numeric expression for the object's top coordinate.
- DisableTrend – Any Boolean expression for disabling the trend window
- DisableNavigation – Any Boolean for disabling the right-click menu
- DisableTooltip – Any Boolean expression to disable the tooltip display

For example, most widgets will display a pop-up menu in response to a right-click.



The menu that appears is under your control via the Navigator module. It is up to you to decide what to include in this module for your custom tags and what each entry should do.

Related Information:

...Navigator Calls (Shortcut Menu)

...ToolTip Contents

...Opening an HDV (PKTrend) Window

...Common Module Example

...Rules for the Common Module

Navigator Calls (Shortcut Menu)

Calls to the Navigator result in a shortcut menu being associated with a tag type. When the user right clicks on the graphic object representing the value of a tag, the shortcut menu opens and enables the user to configure the tag's properties. Tag properties modified using the shortcut menu are automatically written to the tag properties database, following a call to the "Refresh" module.
Access to the shortcut menu is controlled by the Security Manager, which has a specific system bit set aside for this purpose (\SecurityManager\PrivBitConfigure). The Security Manager handles the privileges granted to the logged on user, and disables any shortcut menu options or tag properties for which the logged on user is not authorized.
See: Security Features for Tags.

Related Information:

...Navigator Module Parameters

Navigator Module Parameters

The activation of the Navigator should not occur when a tag's widget is in preview mode or when the Navigator module is called from a container's drawing module.
The first parameter for the Navigator module (Enable), detects if the system is in preview mode, or if the Navigator module is being called by a container module.)
Parameters of the Navigator module:

- **Enable** – The Enable parameter is set to 1 (true) when the shortcut menu may appear. The shortcut menu is disabled during the preview and placement process.
- **Left** – The left coordinate of the object to click upon to open the shortcut menu.
- **Bottom** – The bottom coordinate of the object to click upon to open the shortcut menu.
- **Right** – The right coordinate of the object to click upon to open the shortcut menu.
- **Top** – The top coordinate of the object to click upon to open the shortcut menu.

The Navigator takes groups of four additional significant parameters that describe the menu items and their actions. These parameters are appended to the end of the parameter list in groups of four. One group of four defines one line in the shortcut menu.

- **NavContents** – The shortcut menu option's name. If the name is set equal to "--", a beveled ruling line is placed in the menu, rather than a selectable text option.
- **NavToggles** – A pointer to a value that is a toggle option (i.e. each time this shortcut menu option is selected in the shortcut menu, it is toggled on or off (i.e. is logically inverted)). When the value of the variable pointed to is true, the shortcut menu option is toggled on, and a checkmark appears to its left. When the value of the variable pointed to is false, the shortcut menu option is toggled off, and no check mark appears to its left. Any element may be invalid if not used. If the NavTabNums element is used to open the tag's properties folder, this value must be invalid. If the variable pointed to by this parameter is in a module other than the one pointed to by the "Root" parameter (please see previous section), the NavTabNums parameter contains the object value of that other tag.

  The NavToggles parameter can also serve a very different purpose if it contains an object value, in which case it will be the tag instance to use to launch the TabNums parameter actions within. This can be used to allow direct access to sub-tag parameter configuration folders, or to launch modules within other contexts. This cannot be used in combination with the TabNums

parameter value being invalid, since the TabLabels will not correspond to the correct tag properties folder.

- **NavDisabled** – A status value that enables you to disable any menu item. An invalid entry in this parameter is treated as the item being enabled.
- **NavTabNums** – An index that indicates the tab number (of the tag properties folder) to display if the shortcut menu option is selected (tag properties folder tabs are labeled starting at 0). If this value is invalid, the entire tag properties dialog is opened (i.e. rather than a single tab displayed as a window, the tag properties dialog will open to the ID tab, and will additionally reveal its other tabs).

If this value is a text string, a module by that name is launched within the scope of the calling tag. If the corresponding NavToggles entry has an object value, the module is launched within the scope of that object value.

If the variable pointed to by the NavToggles parameter is in a module other than the one pointed to by the "Root" parameter, the NavTabNums parameter contains the object value of that other tag.

If the NavToggles parameter is a pointer to a parameter of a tag that is not in that current tag, this value must be the object value of the tag where that toggled parameter exists.

To summarize, there are the following modes for these parameters:

Action	NavToggles	NavTabNums
Full tag properties folder	Invalid	Invalid
Single tab tag properties folder	Invalid	Tab number
Toggle parm in this tag	Pointer to the parm	Invalid
Toggle parm in another tag	Pointer to the parm	Object value of other tag
Launch module in tag scope	Invalid	Module name (text)
Launch module in other scope	Scope in which to run	Module name (text)

ToolTip Contents

When the user holds the mouse pointer over a widget, the normal VTScada behavior is to display the associated tag's name and description. You can control what text will be displayed for your tag by including a call to WinToolTipCtrl in your tag's Common module.
If your Common module does not include a call to WinToolTipCtrl, then there won't be a tool-tip.
The target for the tool-tip should be the same as the area parameters passed to the Common module. If this point is contained within another object and the container calls this Draw module, then you do not want this tool tip note to display. This is typically indicated by the calling container's drawing module adding one additional parameter to the list. By detecting that the number of parameters that the Draw is called with is not the same as the number of formal parameters, you can prevent the tool tip note from appearing. This will also prevent the note from appearing during preview and placement which is also desirable.
For example:

```
  WinTooltipCtrl(Left, Bottom, Right, Top,
                 WTTS_FLAG_TRACKINACTIVE + (PickValid(\NoBalloonTips,
0) ?
                 0 : WTTS_FLAG_BALLOON),
                 \Description, \Name, Invalid,
                 \ShowTip && ! IsAPreview && ! \GetUserSession
()\NavActive
                          && PickValid(! DisableTooltip, 1),
                 \TipFont\Value);
```

Opening an HDV (PKTrend) Window

If your tag contains a value that can be trended, then you may include a call to PKTrend in the Common module. This will watch for a left-click within the area of the widget and open a Historical Data Viewer window in response.
This code is generally incompatible with the control dialog code which uses the same mouse button. The first four parameters should match the area as indicated by Left, Bottom, Right & Top because they specify the

target area for the mouse. The last parameter is the object value of this point instance.
This function will automatically be disabled when the page is in editing mode.
For example:

```
  \PkTrend(Left, Bottom, Right, Top, \Root,
           ! IsAPreview && PickValid(! DisableTrend, 1)
            { Disable when called from a container's draw  });
```

Common Module Example

The code to enable this menu for the Analog Status tag is as follows:

```
{======================== AnalogStatus\Common
=========================}
{ This module handles the common actions associated with all drawing
}
{ modules for this point. It will be called by all external drawing
}
{ modules.
}
{=========================================================================-
==}
(
  Left                      { Area occupied by the drawing object
};
  Bottom;
  Right;
  Top;
  DisableTrend              { Flag to disable trend windows
};
  DisableNavigation         { Flag to disable navigator menu
};
  DisableTooltip            { Flag to disable tool tips
};
)

[
  ManualDisabled            { Manual Disabled flag
};
  QuestionableDisabled      { Questionable Disabled flag
};
  AlarmDisabled             { Low  alarm Disabled flag
};
  IsAPreview                { Flag - TRUE if this is deemed a preview
};
]

Common [

   { This variable indicates if the widget calling this common
     module is a preview. The value may be useful for disabling
```

```
certain
     functionality. }
  IsAPreview = IsDrawMethodPreview();


{*********************************************************************-
*}
  { "Post-it" note section
}

{*********************************************************************-
*}

  WinTooltipCtrl(Left, Bottom, Right, Top,
                 WTTS_FLAG_TRACKINACTIVE +
                 (PickValid(\NoBalloonTips, 0) ? 0 : WTTS_FLAG_
BALLOON),
                 \Description, \Name, Invalid,
                 \ShowTip && ! IsAPreview &&
                   ! \GetUserSession()\NavActive &&
                   PickValid(! DisableTooltip, 1),
                 \TipFont\Value);


{*********************************************************************-
*}
  { Navigator menu section
}

{*********************************************************************-
*}

  ManualDisabled       = ! PickValid(\SecurityManager\SecurityCheck
(\SecurityManager\PrivBitManualData,   1), 0);
  QuestionableDisabled = ! PickValid(\SecurityManager\SecurityCheck
(\SecurityManager\PrivBitQuestionable, 1), 0);
  AlarmDisabled        = ! PickValid(\SecurityManager\SecurityCheck
(\SecurityManager\PrivBitAlarmInhibit, 1), 0);
  \Navigator(! IsAPreview && PickValid(! DisableNavigation, 1)
  { Opening condition for the folder },
             Left, Bottom, Right, Top    { Target area for opening -
                                            same as the GUI statement
area },
  { Menu line 1} \HelpLabel,        Invalid,      0,
"HelpLaunch",
  { Menu line 2} "--",              Invalid,      0,
Invalid,
  { Menu line 3} \InhibitLowAlarmLabel, &(\InhibitLo), AlarmDisabled,
Invalid,
  { Menu line 4} \InhibitHighAlarmLabel,&(\InhibitHi), AlarmDisabled,
Invalid,
  { Menu line 5} \AlarmSettingsLabel,  Invalid,      AlarmDisabled,
3,
  { Menu line 6} "--",                 Invalid,      0,
Invalid,
  { Menu line 7} \ManualDataLabel,     Invalid,      ManualDisabled,
```

```
1,
  { Menu line 8} \LoggingDeadbandLabel, Invalid,     ManualDisabled,
1,
  { Menu line 9} \QuestionableLabel,   &(\Questionable), Ques-
tionableDisabled,Invalid,
  { Menu line 10} "--",                Invalid,      0,
Invalid,
  { Menu line 11} \PropertiesLabel,    Invalid,      0,
Invalid);


{*********************************************************************-
*}
  { Trend window pop-up section
}

{*********************************************************************-
*}

  \PkTrend(Left, Bottom, Right, Top, \Root,
           ! IsAPreview &&
           PickValid(\AITrendEnable, 1) &&
           PickValid(! DisableTrend, 1)
            { Disable when called from a container's draw });

]

{ End of AnalogStatus\Common }
```

Rules for the Common Module

- A module called "Common" must be defined within the tag. The Common module handles all of the common drawing object functions that are implemented in the graphic drawing modules. These functions typically include a call to the Navigator module, the ToolTip module, and the PKTrend module, as well as calls to code for control dialog boxes. Any external graphic modules must make a call to the Common module within the tag.
- The Common module takes parameters that define the area of the screen occupied by the drawing object and that enable or disable user interface features. The seven parameters of the Common module are:
  - Left – Any numeric expression for the object's left side coordinate.
  - Bottom – Any numeric expression for the object's bottom coordinate.
  - Right – Any numeric expression for the object's right side coordinate.
  - Top – Any numeric expression for the object's top coordinate.
  - DisableTrend – Any Boolean expression for disabling the trend window

    - DisableNavigation – Any Boolean for disabling the right-click menu
    - DisableTooltip – Any Boolean expression to disable the tooltip display
- Add the call to the "\Navigator" in each of the drawing modules so that when the user right clicks upon the object, the shortcut menu opens and enables the end user to modify the parameters of the selected tag. (Information on the Navigator can be found in Navigator Calls (Shortcut Menu)).

Linking to a Driver

One of the parameters shown in an example at the beginning of this chapter was:

```
DeviceTag  <:TagField("SQL_VARCHAR(255)", "I/O Device Name", 3,
FALSE, "SitePoint", "IODeviceLabel"):> = "*Driver";
```

When a user configures an instance of this tag, the name of the associated I/O device driver will be stored in this parameter.
Rather than using this parameter directly within the tag code, it is common to use the tag's Variables section to specify a location where the object value of the I/O device will be stored. Also, the raw value from the I/O tag will not be used directly as the tag's value – rather, it will be stored in a local variable. When the raw I/O value changes, scaling or other operations can be applied to find the tag's new value.
Updating the example code, the Variables section will now look like the following, where the last line refers to the driver:

```
    { Variables }
  Value (5)             { Scaled value for this point.          };
  RawValue              { Data value read from the IO
};
  Root                  { Set to individual instance of this module
};
  SitePoint             { Object value of IODevice parameter
};
```

The Main state will be responsible for linking the variable to the object value of the I/O device tag:

```
AnalogInMain [
   SitePoint = Scope(Root, DeviceTag);
]
```

The code examples just shown will suffice to link your tag to a I/O device driver. It will be the job of your tag's Refresh module to add or remove reads for the addresses in the device driver in response to any change in parameters. Other modules will take care of handling data sent from the driver.

You must also provide a way for the developer to specify the address to use on the I/O device driver. The AddressEntry and PAddressEntry functions both serve this purpose. These functions will display a standard edit field, or if the driver includes an AddressAssist module, will display the user-interface elements coded there. (see: Providing an AddressAssist Window in the Communication Drivers chapter.)

Related Information:

...Triggering a Data Read

...The NewData Module

...Writing Data: The Set Module

Triggering a Data Read

Having established a link between your tag and a driver tag (previous topic), your Refresh module must issue an AddRead request to the driver in order to start receiving data. If any of the connection parameters change, this request should be cleared and a fresh AddRead issued.

Once the AddRead has been issued to the driver instance, it is then up to the driver code to poll the PLC at the ScanInterval rate and send new values back to the specified destination. Note: The Refresh module is not used to handle new values from the driver. For details on how the driver sends values to your tag, see The NewData Module.

The following example (taken from the Analog Input tag's Refresh module) shows how the AddRead function should be called. The tag's RawValue parameter is used as the destination for the value read from the driver. You may then write code that scales this before using the result as the tag's value or, if the manual data flag is set, ignores the raw data value in favor of the manual data value. Note that this example is

for analog data – digital values are read using the same technique, but would not be scaled.

```
Refresh [
  If 1;
  [

    {***** Scan Interval *****}
    ScanInterval = PickValid(Cast(ScanInterval, 3),
                             GetDefaultValue(&ScanInterval));

    {***** The connection parameters were set or changed *****}
    IfThen (PickValid(Parm[#IODevice]      != DeviceTag,    1) ||
            PickValid(Parm[#ScanInterval]  != ScanInterval, 1) ||
            PickValid(Parm[#Address], "")  != PickValid(Address, ""),

      {***** Remove the previous read request (this will do nothing
             if the parameters are invalid) *****}
      Scope(Root, Parm[#IODevice])\Driver\DelRead(Parm[#Address],
                                      &RawValue, Parm[#ScanIn-
terval]);

      RawValue = Invalid { old value no longer valid };

      {***** Add a new read request *****}
      Scope(Root, DeviceTag)\Driver\AddRead(Address
                        { Address in the PLC for the data },
                                              1
                        { Number of data elements to get         },
                                              &RawValue
                        { Destination pointer for data read      },
                                              ScanInterval
                        { Scan Interval                          });
    );

{*********************************************************************}
    { ManualValue
}

{*********************************************************************}
    ManualValue = Cast(ManualValue, 3 { Float });
    IfElse (Valid(ManualValue),
      Value = ManualValue;
    { Else } IfThen (!ExternalValue,
      Value = IsText ? RawValue : Scale(RawValue, UnscaledMin,
                                        UnscaledMax, ScaledMin,
ScaledMax);
    ));
```

The NewData Module

While it is true that there is more than one method for reading values from an I/O driver, use of the NewData module is recommended for all new tags.

If your tag includes a module named NewData, with the structure similar to that shown in the following example, the I/O driver will use it to send values from the device to the tag.
NewData should have the following five parameters where the last two, Quality and ServiceSync, are optional.

```
NewData
(
  Addr              { Address we asked for                          };
  TimeStamps        { Single value or 1D array of UTC timestamps  };
  Data              { Single value or 1D array of unscaled data
                      values                                        };
  Quality           { Single value or 1D array of Quality data    };
  ServiceSync       { used to denote that the NewData call was made
as a result of a
                      server synchronization (the server has just
started up and this
                      data has been taken from a running server) };
)
```

For purposes of comparison, the body of the NewData Modules from both the Analog Status tag and the Digital Status tag are reprinted here. Items to note are:

- The RawValue is found in the first item of the Data array.
- The NeedValueUpdate variable scopes to the parent tag module. This is used to alert the tag to the fact that a new value has been received and that the tag's value should be updated.
- The largest portion of the module provides support for drivers that send history values. These are scaled and logged as required.

NewData() example for reading analog values:

```
<
{========================= NewData ==============================}
{ Subroutine that receives and processes incoming data from driver }
{ RefreshData in VTSDriver guarantees that TimeStamps is a valid   }
{ time stamp or an array of valid time stamps regardless of the    }
{ validity of Data                                                 }
{====================================================================}
NewData
 (
  Addr              { Address we asked for                          };
  TimeStamps        { Single value or 1D array of UTC timestamps  };
  Data              { Single value or 1D array of unscaled data
                      values                                        };
)
 [
  I                 { General counting variable                     };
  HistSizeRet       { Size of history list returned                 };
```

```
  MaxHistTS          { Highest TS in history list                  };
  LastHistPos        { Index of newest data in history list        };
  HistoryValues      { Scaled history array to pass to data logger };
  HistImgTimeStamps  { For history use, 1D array, image of
                       TimeStamps                                  };
  HistImgData        { For history use, 1D array, image of
                        Data                                      };
  NeedValueUpdate = FALSE   { TRUE if the tag needs to recalculate
                              Value                               };
]

Main [
  If 1;
  [
    IfElse (PickValid(Addr == CurrValAddr, 0), Execute(
       { Current value data }
      RawValue = Data[0];
      RawTS    = PickValid(TimeStamps[0], CurrentTime(1));
      NeedValueUpdate = TRUE;
    );
    { Else } IfThen (PickValid(Addr == HistoryAddr, 0),
   { History data might have been Recv'd as single value or as an
array }
   { Size of history array }
      IfElse(Valid(HistSizeRet = ArraySize(Data, 0)), Execute(
         { Data is an array }
        HistImgTimeStamps = TimeStamps;
        HistImgData       = Data;
      );
      { Else } Execute(
         { Data is a single value so copy to a single-value array }
        HistSizeRet          = 1;
        HistImgTimeStamps    = New(HistSizeRet);
        HistImgData          = New(HistSizeRet);
        HistImgTimeStamps[0] = TimeStamps;
        HistImgData[0]       = Data;
      ));

       { If the current tag value comes from history }
      IfThen (ValueFromHist,
         { Extract RawValue & RawTS from history list if no
           Current Value address defined }
         { Find the newest history TS }
        MaxHistTS = AMax(HistImgTimeStamps[0], HistSizeRet);
         { Only use it if greater than current value TS }
        IfThen (MaxHistTS > PickValid(TimeStamp, 0),
           { Where is this in the array ? }
          LastHistPos = Lookup(HistImgTimeStamps[0],
                               HistSizeRet, MaxHistTS);
           { Set our values to the newest history value }
          RawValue = HistImgData[LastHistPos];
           { Set the time as well }
          RawTS = HistImgTimeStamps[LastHistPos];
          NeedValueUpdate = TRUE;
        );
      );
```

```
      { Scale the data & log it }
     HistoryValues = New(HistSizeRet);
     I = 0;
     WhileLoop(I < HistSizeRet,
       HistoryValues[I] = Scale(HistImgData[I], UnscaledMin,
                                UnscaledMax, ScaledMin, ScaledMax);
       I++;
     );

      { Log history values }
     \HistorianManager\WriteHistory(Root, HistImgTimeStamps,
                                    HistoryValues);
   ));

   IfThen(NeedValueUpdate && !Valid(ManualValue),
     UpdateValue(ValueFromHist);
{ The parameter tells UpdateValue to not log Value if it came
          from a history value; this has been logged already }
   );

   Return(0);
  ]
]

{ End of AnalogStatus\NewData }
>
```

NewData() example for reading digital values:

```
<
{======================== NewData
================================}
{ Subroutine that receives and processes incoming channel data from
}
{ driver RefreshData in VTSDriver guarantees that TimeStamps is a
}
{ valid time stamp or an array of valid time stamps regardless of
}
{ the validity of Data
}
{=================================================================-
=}
NewData
(
  Addr                 { Address we asked for (not used)
};
  TimeStamps           { Single value or 1D array of timestamps
};
  Data                 { Single value or 1D array of unscaled data
                         values
};
  Quality              { Single value or 1D array of Quality data
};
)

[
  HistSizeRet          { Size of history list returned
```

```
};
  MaxHistTS            { Highest TS in history list
};
  LastHistPos          { Index of newest data in history list
};
  HistoryValues        { Scaled history array to pass to data logger
};
  HistImgTimeStamps    { For history use, 1D array, image of
                         TimeStamps
};
  HistImgData          { For history use, 1D array, image of
                         Data
};
]

Main [

  If 1;
  [
    IfElse (PickValid(Addr == Bit0Address, 0), Execute(
       { If b0 address }
      RawValue0 = Data[0];
      RawTS     = PickValid(TimeStamps[0] - TimeZone(0), CurrentTime
());
    );
    { Else - b1 address } IfElse(PickValid(Addr == b1ValAddr, 0),
Execute(
      RawValue1 = Data[0];
      RawTS     = PickValid(TimeStamps[0] - TimeZone(0), CurrentTime
());
    );
    { Else } IfThen (PickValid(Addr == HistoryAddr, 0), { History
Address }
     { History data might have been Recvd as single value or as an
array }
     { Size of history array }
      IfElse(Valid(HistSizeRet = ArraySize(Data, 0)), Execute(
         { Data is an array }
        HistImgTimeStamps = TimeStamps;
        HistImgData       = Data;
      );
      { Else } Execute(
         { Data is a single value so copy to a single-value array }
        HistSizeRet          = 1;
        HistImgTimeStamps    = New(HistSizeRet);
        HistImgData          = New(HistSizeRet);
        HistImgTimeStamps[0] = TimeStamps;
        HistImgData[0]       = Data;
      ));

       { If the current tag value comes from history }
      IfThen (ValueFromHist,
         { Extract RawValue & RawTS from history list if required }
         { Find the highest history TS }
        MaxHistTS = AMax(HistImgTimeStamps[0], HistSizeRet);
         { Only use it if greater than current value TS }
        IfThen (MaxHistTS - TimeZone(0) > PickValid(TimeStamp, 0),
```

```
          { Where is this in the array ? }
         LastHistPos = Lookup(HistImgTimeStamps[0], HistSizeRet,
                              MaxHistTS);
          { Set our value to the newest history value }
         RawValue0 = HistImgData[LastHistPos];
         RawTS     = HistImgTimeStamps[LastHistPos] - TimeZone(0);
       );
     );

      { Invert the data & log it  }
     HistoryValues = New(HistSizeRet);
      { Make a copy of the data array. }
     ArrayOp2(HistoryValues[0], HistImgData[0], HistSizeRet, 0);

      { Some data massage required... }
      { Force all data to 0/1 values.
         This first step actually returns the inverted data. }
     ArrayOp1(HistoryValues[0], HistSizeRet, 0, 12 {==});
     IfThen (!InvertInput,
        { Invert the data back to normal if not inverted }
       ArrayOp1(HistoryValues[0], HistSizeRet, 0, 12 {==});
     );

      { Log history values }
     \HistorianManager\WriteHistory(Root, HistImgTimeStamps,
                                    HistoryValues);
   )));

   Return(0);
  ]
]
{ End of NewData }
>
```

Writing Data: The Set Module

Tags that write data to a driver tag must have a subroutine module named "Set". This module will perform 3 actions:

- Check that the value to be written is valid. Do not write an invalid value.
- Scale or invert the value for the PLC according to the tag's scaling parameters.
- Ensure that the value is sent to all workstations in the network with a call to the RPCManager's Send function.

The return value from the Write() function will be the object value of the write module launched. When this value becomes invalid, the caller can assume that the write is complete. Note that, for a client machine on a network, this value will always be invalid.

The following example is taken from the Analog Control tag:

```
<
{======================== AnalogOutput\Set
============================}
{ This subroutine writes a value to SitePoint. The parameter is the
}
{ value to write, which may be inverted before writing.         }
{
}
{=================================================================-
=}
Set
(
  NewValue              { Value to write to the SitePoint
};
)

[
  WriteObj              { Object value of launched write
};
  WriteValue            { Actual value written
};
]

Set [
  If 1;
  [
    NewValue = PickValid(Limit(NewValue, ScaledMin, ScaledMax),
NewValue);
    SaveValue = Value = NewValue;

    {**** Don't bother to write to the PLC if the value is invalid
*****}
    IfThen (Valid(Value),
      WriteValue = PickValid(Scale(NewValue, ScaledMin, ScaledMax,
                                   UnscaledMin, UnscaledMax),
NewValue);
      WriteObj   = SitePoint\Driver\Write(Address, 1, &WriteValue);
    );

    {***** Send the value to everybody *****}
    \RPCManager\Send(SitePoint\Driver\RPCService { service },
                     \LocalGUID { GUID },\RPC_ACCEPT_FILTER {mode cut-
off },
                     1 { server }, Invalid { machine }, 1 { clients },
                     0 { locally }, 1 { recursive }, "SetValue" { mod-
ule },
                     "RPCManager" { scope  }, Root { queue data },
                     Invalid { InputSessionID },
                     { Parameters: } \LocalGUID, Concat("VTSDB\",
Name),
                     "Value", Value);

    Return(WriteObj);
  ]
]
{ End of AnalogOutput\Set }
>
```

Make a Custom Tag Visible to OPC Clients

When running VTScada as an OPC server, the values of standard tags are available to clients. This will not be true of your custom tags unless you add modules that specify what will be made available.
The following modules may be added. Examples of these follow.

OPCGetTagAttributes	This tells the VTScada OPC server whether this tag can be read from or written to, as well as the data type it is to return. Required for the tag to be visible to an OPC client.
OPCGetTagProperties	Optional. Returns a dictionary of all the items that will be available from the tag.
OPCReadTagValue	Must be included if the tag is to be readable.
OPCWriteTagValue	Only required if the tag is writeable.

For any OPC constants that are not defined locally, you should be able to add a backslash in order to obtain it from VTScada.
UpdateOPCData() is a VTScada module. As with the constants, if it is not in the immediate scope, you can add the backslash operator.
example:

```
IfThen(Valid(\OPCServerHandle),
   \UpdateOPCData(Self(), \OPC_PROP_RAW_VALUE, RawValue);
);
```

This will tell the server code to update the RawValue attribute, since it has changed. OPCServerHandle is set higher up the scope tree when the OPC server is enabled – this prevents that code from being called if it is not present.
Make the tag's value available to OPC clients. If included, your tag's main state should call \UpdateOPCData(Self()); whenever the value changes.

```
<
{============================ OPCReadTagValue
==============================}
{ Subroutine that returns the current value, quality, and timestamp
of this }
{ tag for OPC purposes.                  }
```

```
{=========================================================================-
=======}
OPCReadTagValue
(
  pValue             { (Output) The value of the tag, must be a Valid
value };
  pQuality           { (Output) The quality of the value
};
  pTimestamp         { (Output) The timestamp of the value
};
  pTimestampIsUTC { (Output) Indicates whether the timestamp is in
UTC   };
)

Main [
  If 1;
  [
    *pValue     = PickValid(Value, Valid(Cast(Value, \#VTypeText)) ?
"" : 0);
    *pTimestamp = Timestamp - TimeZone(0);
    *pQuality = Operational
                ? (Operable && Valid(ManValue))
                  ? \OPC_QUALITY_GOOD_LOCAL_OVERRIDE
                  : Valid(Value)
                    ? \OPC_QUALITY_GOOD
                    : \OPC_QUALITY_BAD
                : \OPC_QUALITY_UNCERTAIN;
    Return(Invalid);
  ]
]

{ End of OPCReadTagValue }
>
```

This will be called when an OPC client is requesting the values. The parameter, pValue, is what you want it to show, normally gained from Value The following module example creates a dictionary of properties that will be available to OPC clients. Remove any properties that you do not intent to make available. For example, if you don't need OPC_PROP_ DESCRIPTION declared then remove the line within this module. A complete list of OPC values can be found in Properties of Tag OPC Items.

```
<
{======================= OPCGetTagProperties
============================}
{ Subroutine that returns a dictionary of OPC properties and values
}
{ supported by this tag. The dictionary is keyed by property ID.
}
{=========================================================================-
=======}
OPCGetTagProperties
 [
```

```
  PropDict;
]
Main [
  If 1;
  [
    PropDict = Dictionary();
    PropDict[OPC_PROP_EU_UNITS]                = Units;
    PropDict[OPC_PROP_DESCRIPTION]             = Description;
    PropDict[OPC_PROP_HIGH_EU]                 = ScaledMax;
    PropDict[OPC_PROP_LOW_EU]                  = ScaledMin;
    PropDict[OPC_PROP_HIGH_INSTRUMENT]         = UnscaledMax;
    PropDict[OPC_PROP_LOW_INSTRUMENT]          = UnscaledMin;
    PropDict[OPC_PROP_TIMEZONE_MINUTE_OFFSET]  = TimeZone(0)/60;
    PropDict[OPC_PROP_AREA]                    = Area;
    PropDict[OPC_PROP_DEVICE_TAG]              = DeviceTag;
    PropDict[OPC_PROP_ADDRESS]                 = Address;
    PropDict[OPC_PROP_RAW_VALUE]               = RawValue;
    PropDict[OPC_PROP_LOW_ALARM]               = Cast(AlarmLo,3); {
In case AlarmLo is a tag or expression }
    PropDict[OPC_PROP_HIGH_ALARM]              = Cast(AlarmHi,3); {
In case AlarmHi is a tag or expression }
    Return(PropDict);
  ]
]

{ End of OPCGetTagProperties }
>
```

Example to define whether the tag can be read from or written to by the OPC client:

```
<
{======================= OPCGetTagAttributes
==============================}
{ Subroutine that returns the OPC attributes of this tag.
}
{=========================================================================-
=======}
OPCGetTagAttributes
 (
  pAccessRights      { (Output) The read/writeability of the tag
};
  pDataType          { (Output) The COM datatype of the value of the
tag    };
)

Main [
  If 1;
  [
    *pAccessRights = Output ? \OPC_ACCESS_READWRITEABLE : \OPC_
ACCESS_READABLE;
    *pDataType = Valid(Value)
                ? Valid(Cast(Value, \#VTypeText))
                  ? \VT_BSTR
                  : \VT_R8
                : \VT_EMPTY;
    Return(Invalid);
```

```
  ]
]

{ End of OPCGetTagAttributes }
>
```

pAccessRights defines whether the tag's value can be read or written to according to the assignment of one of the following constants:

```
 CONSTANT OPC_ACCESS_READABLE        = 1;
 CONSTANT OPC_ACCESS_WRITEABLE         = 2;
 CONSTANT OPC_ACCESS_READWRITEABLE  = 3;
```

pDataType sets the data type of the tag to one of the following for compatibility with component object model standards:

```
CONSTANT VT_EMPTY= 0;
 CONSTANT VT_NULL= 1;
 CONSTANT VT_I2= 2;
 CONSTANT VT_I4= 3;
 CONSTANT VT_R4= 4;
 CONSTANT VT_R8= 5;
 CONSTANT VT_CY= 6;
 CONSTANT VT_DATE= 7;
 CONSTANT VT_BSTR= 8;
 CONSTANT VT_BOOL= 11;
 CONSTANT VT_VARIANT= 12
```

If your tag will allow OPC Write access, you will need a module similar to the following example. Note that this example does not provide for any checking of the request – something may wish to add.

```
<
{========================== OPCWriteTagValue
============================}
{ Subroutine called when an OPC client wants to write a value to this
tag. }
{=========================================================================-
=======}
OPCWriteTagValue
 (
  NewValue         { (Input) The value to be written, must be a Valid
value };
)

Main [
  If 1;
```

```
  [
    Set(NewValue);
    Return(0);
  ]
]

{ End of AnalogOutput\OPCWriteTagValue }
>
```

For a description of the Set() module, see: Writing Data: The Set Module.

Logging Tag Data

A logger tag can be attached to any tag that:

- Is part of the group, Numerics (See: Tag Groups)
- Exposes a variable, "Value," which is numeric. (variable classes 1 – 6. See: Required Variables for a list of class definitions.)

The simplest way to log your tag's data is to ensure that it satisfies the above two conditions and then configure a Logger tag for it. For configuration details, see: Configure a Tag for Logging.

Note that all variables in your tag module which are declared with a class constant from 1 to 6 will be logged.

Some VTScada tags, such as the Analog Status, have logging functionality built in. This requires extra work on your part, but the benefit is that you can configure logging to work exactly the way you want, and can reduce your application's tag count by not requiring an additional Logger tag.

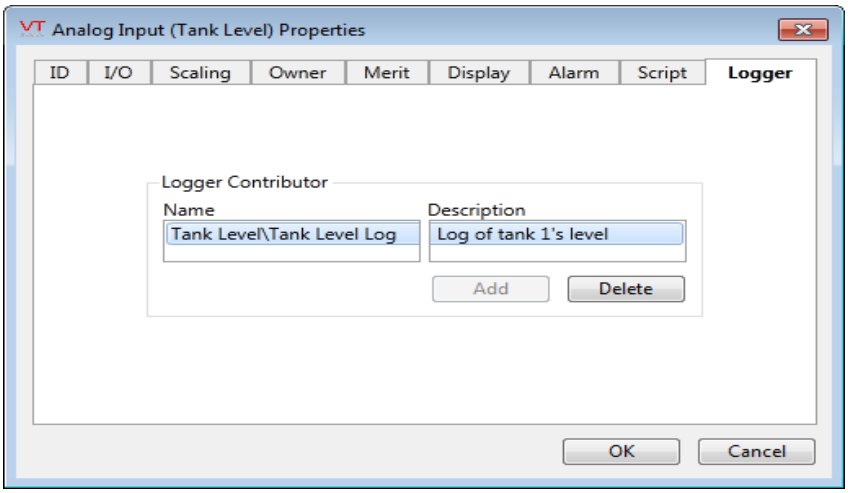
Related Information:

...Configure a Tag for Logging

...Custom Logging for Tags

Configure a Tag for Logging

A tag need only be a member of the Numerics group and expose a numeric variable in order to be used as the data source for a Logger tag. It is common practice to include a Logger tab in a tag's configuration dialog.



To do this, include the variable "LogContributors" in your tag definition module and add the following to your tag's configuration panel code:
Variable declarations:

```
  LogContribs             { List of log contributors
};
  NumLogs                 { Number of log contributors
};
```

Logger panel:

```
Logger [
  If Current != x Switch;

  {***** Logger contributor list *****}
  GUITransform(80, 203, 420, 96,
               1, 1, 1, 1, 1            { No scaling
},
               0, 0, 1, 0               { No movement; visible;
reserved },
               0, 0, 0                  { Not selectable
},
               \DialogLibrary\PContributor("LogContributors" { holder
},
               \Root,
               Parms[\#ContributionType] { contrib type },
               "LogPoint" { point type },
               \LoggerContribLabel, 1 { & 2 - ID },
               NumLogs ? 1 : 0 { no add if already 1 }));

  {***** Keep track of the number of log contributors *****}
  LogContribs = Scope(\Root, "LogContributors");
  NumLogs     = PickValid(ArraySize(*LogContribs, 1), 0);
]
```

Custom Logging for Tags

One reason to add logging code to your tag modules is to free the user from doing any configuration work to enable data logging. It also provides you with control over what is logged and when.
Three things are required in order for your tag to do its own logging:

- The tag must have a parameter or a variable named "HistorianName" which will hold the name of the Historian tag (the Historian tag controls where the data is stored).
- The variables to be logged must be declared using one of the class constants 1 through 6 (see following table).

Class Constant	Value Type
Class 1	Bit
Class 2	Unsigned byte
Class 3	16-bit integer
Class 4	32-bit integer
Class 5	Double precision floating point
Class 6	Text

- You must call \HistorianManager\WriteHistory(Root) each time you want to write log data.

In general, it is better to use a parameter than a variable for HistorianName. This will provide greater flexibility in configuration. For example, in the parameters declaration of the Analog Status tag, HistorianName is declared as follows. Note that the System Historian is provided as the default value.

```
  HistorianName  <:TagField("SQL_VARCHAR(255)" ):> = #SYSTEM_
HISTORIAN
                                        { Historian Tag name
};
```

The call to write to the historian should be placed such that it is called whenever the value changes sufficiently to merit a new record (a dead-band parameter is commonly used to avoid logging of very small changes). The call may also be placed in the tag's Refresh module (or a

submodule called from Refresh) for the case where the HistorianName parameter changes, in order to log invalid to the old Historian and the current value to the new one.
If your tag supports the reading of historical data that has been logged on the PC, you will also need to loop through the arrays in NewData.
The basic form of the code will be as follows:

```
\HistorianManager\WriteHistory(Root, TimeStamp, Value);
```

Where:

- "Root" is the root value of the tag
- "TimeStamp" is a UTC timestamp that you should collect just before writing
- "Value" contains the data to be recorded.

For complete details about using WriteHistory, refer to the VTScada Function Reference, WriteHistory.
You may choose to record any calculated values you would like, and may impose conditions on whether or not the write occurs with each refresh. For example, the Analog Status tag will not perform a write until the tag's value has changed from the previously written value by at least a threshold amount (controlled by the parameter, Deadband, if set).

```
IfThen(PickValid(Abs(Value - LastLoggedValue) >= Threshold, 1),
  \HistorianManager\WriteHistory(Root, TimeStamp, Value);
  LastLoggedValue = Value;
);
```

Note that, any tag with a HistorianName parameter or variable will be added to both the "Loggers" and the "Trenders" tag groups, automatically.

Related Information:

...Data Logged or Trended Variables in Tag Modules

Upgrading Tags That Used LogManager or Logger

Prior to the release of VTS 10, logging was done using the LogManager module. This is now obsolete, having been replaced by the HistorianManager.

Any references to LogManager or Logger in older tags must be updated to use the HistorianManager API instead. See: Historian Manager.

Adding Alarms to Custom Tags

The VTScada Alarm tag can monitor the value of any other tag that is a member of the group Numerics and that has a Value variable. Provided that your custom tag has these two features, it can be used to trigger an alarm.

In addition to the tag features just mentioned, you should also ensure that when the application starts, the VTScada AlarmManager has a chance to start before your tag is initialized. This is commonly done in the tag's initialization state, along with ensuring that the expression manager has started:

```
MyTagInit [
  If \AlarmManager\Started && \ExpressionManager\Started MyTagMain;
  [
    CriticalSection(
      Root = Self
      Refresh();
    );
  ]
]
```

If you would like your custom tag to be aware of the Alarm tags that use it as a set point, then it is necessary to configure Container variables in your tag for those alarms. Reasons for doing so include:

- Custom widgets for your tag may need to display the current alarm state.
- Your tag may be a contributor to another tag and should therefore pass its alarm state to that owner.
- You may choose to display a list of attached alarm tags in your custom tag's configuration panel.

To configure alarm containers in your custom tag see: Alarm Containers. For more information about container variables in general, see: Containers and Contributors.

You may decide to add alarm management functions directly to your custom tag, in addition to relying on Alarm tags. For example, both the Analog Status and Digital Status tags have alarm functionality built in. This will require somewhat more code in your custom tag, but you can reduce the number of tags required in your application. For details, see: Adding Built-in Alarms to a Tag.

Related Information:

...Alarm Containers

...Adding Built-in Alarms to a Tag

Alarm Containers

In order for attached Alarm tags to contribute their values and alarm states to your custom tag, you must provide a variable to which they can connect. For example, the Analog Status tag does this as follows.
In the (PLUGINS) section of the variables declaration, the ContributorAdded and the ContributorDeleted modules are declared: (This example shows the complete PLUGINS section from the Analog Status, including the ConfigFolder and Common modules

```
  [ (PLUGINS)
    Shared ConfigFolder       = "AnalogStatusConfig";
    Shared Common             = "AnalogStatusCommon";
    Shared ContributorAdded   = "ContributorAdded";
    Shared ContributorDeleted = "ContributorDeleted";
  ]
```

The following variables are then declared:

```
  { List of variables handling points contributing to this one }
AlarmActive       { This is an array of alarm priorities for all
                    active alarm contributors. Any alarm
                    contributors which are not active will set
                    their element to invalid. };
AlarmUnacked      { This is an array of alarm priorities for all
                    unacknowledged alarm contributors. Any
                    acknowledged alarm will set its element to
                    invalid. This array is a 1 to 1
                    correspondence to the AlarmActive array. };
AlarmContributors { This is the "handle" used by the VTScada code
                    to maintain a list of the alarm contributors };
AlmSetPointPtrs   { This is an array of Pointers to the setpoints
                    of each of the alarm contributors };
AlmPriorityPtrs   { This is an array of Pointers to the
                    Priorities of each of the alarm contributors };
```

The names must appear in your code exactly as shown in this example. As noted in the comments, VTScada will look for and use these variables for the alarm contributors.

Adding Built-in Alarms to a Tag

To add one or more built-in alarms to a custom tag, you will need the following parts:

- Configuration parameters for storing alarm properties. Your tag's configuration panel should include a tab for the alarm properties.
- Local variables and constants for handling your alarms.
- A set of statements in the tag's Refresh module to commission the alarm, setting or updating parameters as needed.
  In the event that the tag is being deleted, decommission should be called.
- A call to \AlarmManager\EvaluateAlarm() to reevaluate the alarm's state with the tag's current value each time that value changes.

Note: Prior to VTScada version 11.2, there was a requirement that separate submodules be created for each alarm if the tag included more than one. This is no longer the case. The GetAlarmObjVal module is now obsolete, but may be maintained for backward-compatibility.

Configuration Parameters and Variables

Alarm records have the configuration fields shown in the following table. Not all need to be set, so select those that are relevant to your tag and application. For example, trips should not be used with an analog tag. Deadbands are not used with digital tags.
At a minimum for a properly functioning alarm, you will need Name, Priority, Function, and Setpoint.
FriendlyName, Area, Description and Disable may not be required, but are highly recommended, and are usually set using properties of the tag.

ConfigurationStruct { All Boolean flags default to FALSE }	
Name	Unique name for the alarm
FriendlyName	Display name of the alarm's source

ConfigurationStruct { All Boolean flags default to FALSE }	
Area	Area
Description	Description. Was "Message" prior to 11.2
Priority	Priority. Must be valid to be commissioned. Must be an integer corresponding to the Alarm Priority tag values.
Reserved	
Disable	TRUE to disable the alarm
DisableParmName	Name of the tag's disable parm. Allows us to get the operator name who made the config change.
OnDelay	Seconds to delay before activating
OffDelay	Seconds to delay before clearing
RearmDelay	Seconds to delay before rearming after ack
Setpoint	Setpoint of alarm evaluation
ValueLabels	Array of labels to display instead of Value or Setpoint. Rarely used by tags other than digitals.
Units	Setpoint units
Function	Enumerated function for alarm evaluation (1)
AlarmType	String identifying the type of alarm
Trip	TRUE if alarm only becomes unacked not active
NormalTrip	TRUE if alarm becomes unacked when it clears
OffNormal	TRUE if alarm only becomes active not unacked
Deadband	Setpoint deadband
PopupEnable	TRUE to enable popup display of active alarm
SoundFile	Filename relative to app path of custom sound
Custom	Array/Dictionary/Structure of custom fields
AdHoc	TRUE if alarm is ad hoc

An example, showing how to configure these as tag properties:

```
SetPoint  <:TagField("SQL_DOUBLE", "Set Point"):> = 0  { State or set-
point that triggers the alarm };
Function  <:TagField("SQL_VARCHAR(255)", "Function Code"):> { Text
string defines the comparison between value and setpoint };
Priority  <:TagField("SQL_DOUBLE", "Priority")   :>      { Numeric
```

```
Priority of this alarm };
Disable   <:TagField("SQL_LONGVARCHAR", "Disable"):> = FALSE { TRUE
to Disable the alarm };
{ ... etc. ... }
```

If your alarm is to be used only for low levels or high levels, you can set the Function field as a constant, otherwise, you will need to create a configuration parameter for it and provide a selection in the user interface. See: Alarm Manager Function Constants.

If creating more than one alarm, create a set of properties for each. For example, SetPointLo and SetPointHi.

As with other tag parameters, a constant should be defined for each alarm parameter. For example, if the parameter constant before SetPoint was 7, then:

```
Constant #SetPoint = 8;
Constant #Function = 9;
```

... and so on.

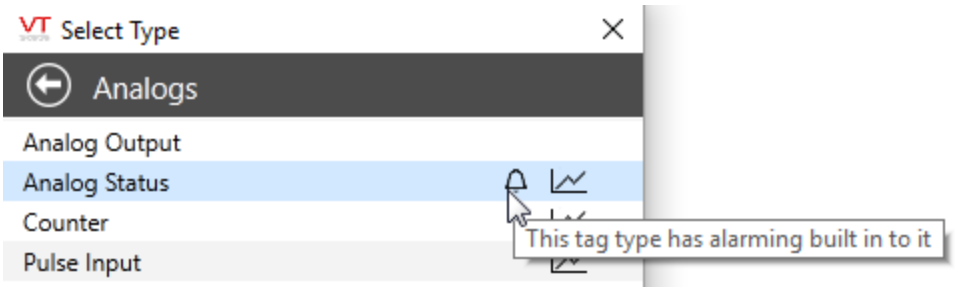
An alarm configuration structure will be required in order to commission your alarm(s). Declare each as a protected variable:

```
PROTECTED AlarmCfg     { Structure of alarm configuration parameters
};
```

Your tag may also need variables to hold any alarm status information that you care to examine during the normal running of the tag and any configuration fields that are to be set but are not being exposed as user-configurable tag properties. The AlarmStatus structure (any name may be used) is typically set immediately after the alarm is commissioned. An example is provided later in this topic.

```
{ Alarm variables }
PROTECTED AlarmStatus  { Structure of alarm status information. };
{ ... other variables as required ... }
```

If the tag has a built-in alarm, that fact should be reflected as an icon in the Tag Browser.



To achieve this, all that is required is to add the following constant declaration. The Tag Browser will take care of the rest.

```
Constant BuiltInAlarm = TRUE { Flag - TRUE if this tag has a built in
alarm };
```

Tag Initialization

Your module should ensure that the Alarm Manager has started before attempting to commission itself. Since it is also typical for a tag to ensure that the Expression Manager has started, that is also shown in the following example.

```
MyTagInit [
  If \AlarmManager\Started && \ExpressionManager\Started MyTagMain;
  [
    CriticalSection(
      Root = Self();
      Refresh();
    );
  ]
]
```

Main State - Handle changes to the Disable parameter

Under the ISA18.2 standard, it is recommended that you provide a way for the tag to be disabled and re-enabled dynamically. This can be done with the tag's Common module (right-click menu), and by allowing the Disable parameter to be configured via a constant, expression or tag (i.e. via a PTypeToggle in the configuration panel).

Changes to this parameter outside the properties dialog are handled in your tag's main state. The configuration object must be updated and the Alarm Manager's Commission function called.

```
AlarmDisable = PickValid(Cast(ValueType(Disable) == \#VTypeObject
                              ? Disable\Value
                              : (Valid(Scope(Root, Disable)\Name)
                                 ? Scope(Root, Disable)\Value
```

```
                                    : Disable),
                                \#VTypeStatus),
                            AlarmDisable,
                            FALSE);
{ Enable/Disable }
If Watch(1, AlarmDisable);
[
  AlarmCfg = \AlarmManager\GetAlarmConfiguration(Root\UniqueID);
  AlarmCfg\Disable = AlarmDisable;
  \AlarmManager\Commission(Root, AlarmCfg, Value);
]
```

Note: You might prefer to set the 5th parameter of Commission to TRUE for these types of modification to the alarm so that the change is not added to the history list.

Refresh Submodule: Configure, Commission and Update

Alarm parameters are handled much like others in the refresh module. (Note: you are advised to use a PTypeToggle in your tag's configuration panel for the disable parameter so that it can be tied to a tag or expression. Other parameters may be configured as you see fit. The following example reflects this detail.)

```
\ExpressionManager\SafeRefresh(&Disable, Parms[#Disable]);
```

```
AlarmRearmEnable = PickValid(Cast(AlarmRearmEnable, \#VTypeStatus),
GetDefaultValue(&AlarmRearmEnable));
```

The Refresh module must call the Commission function of the Alarm Manager, handing it all of the properties you are configuring in your alarm. If the tag contains only one alarm, then the name will be the same as the tag's unique ID value. If there are multiple alarms within the tag, then each will be the UniqueID concatenated with the alarm separator string (defaulting to :#:) followed by a digit starting with 0. Thus, when assigning the alarm name for a single alarm, the code will be:

```
AlarmCfg\Name = Root\UniqueID;
```

When assigning the alarm name for the first of several alarms within a tag, the code will become:

```
AlarmLoCfg\Name = Concat(Root\UniqueID, PickValid(\AlarmSe-
paratorString, ":#:"), "0");
```

Noting that the digit must be incremented for each subsequent alarm.

The name will be used several times, therefore it makes sense to store it in a variable such as "AlarmName" or "AlarmLoName".
The following example shows slightly more than a minimal configuration. Your code will likely set additional properties in the configuration structure.

```
IfElse(Valid(Name), Execute(
  AlarmName               = Root\UniqueID;
  AlarmCfg                 = \AlarmManager\GetAlarmConfiguration
(AlarmName);
  AlarmCfg\Name            = AlarmName;
  AlarmCfg\FriendlyName    = Name      { The tag's short name };
  AlarmCfg\Area            = Area      { The tag's area };
  AlarmCfg\Description     = Concat(PickValid(Concat(\Description, "
"), ""), "Low-level alarm");
  AlarmCfg\Priority        = Priority;
  AlarmCfg\Function        = \AlarmManager\ALM_FUNC_LESS_THAN; { a
low alarm, no user-selection }
  AlarmCfg\Setpoint        = PickValid(Cast(Setpoint, \#VTypeDouble),
AlarmCfg\Setpoint);
  \AlarmManager\Commission(Root, AlarmCfg, Value);
  { ... repeat for more alarms ... }
);
{ Else, the tag is being deleted... }
  IfThen(Valid(Parms[#Name]),
    \AlarmManager\Decommission(AlarmName);
  );
);
```

If your tag needs to know its current alarm state, you should follow this by obtaining a reference to your alarm's entry in the master database. Typically, this is required only for widgets that are able to indicate that state. The reference to the database entry will allow quick access the alarm's current state without having to make function calls.

```
AlarmStatus = \AlarmManager\GetAlarmStatus(Root\UniqueID);
```

Evaluate the Alarm

The preceding should result in a commissioned alarm record when you compile your tag and generate an instance of it in your running application, setting priority to an integer. But, that is not enough to trip or activate an alarm event. You must also call the Alarm Manager's EvaluateAlarm function whenever the value of your tag changes so that the new value can be compared to the setpoint using the assigned comparison function.

If your tag links to a driver and has a NewData submodule, then place the following code there. Otherwise, it might go into the main state, if that is where new values are being calculated. Note that EvaluateAlarm cannot be called in Steady State.

```
If Watch(1, Value);
[
  \AlarmManager\EvaluateAlarm(Root\UniqueID, Value);
```

If the time is being collected from remote equipment with the value, then you should include that in the call as a UTC timestamp:

```
\AlarmManager\EvaluateAlarm(Root\UniqueID, Value, TimeOfValue);
```

Related Information:

...Alarm Manager Function Constants – Enumeration of function constants.

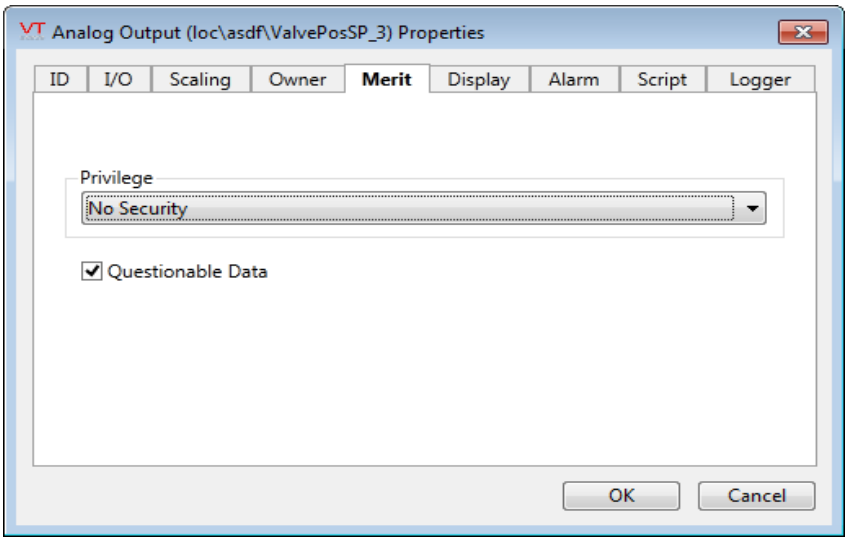
...Alarm Status Structure – Fields in the alarm status structure.

...Alarm Functions – Function list

...Alarm Tab Notes – Displaying a selection of comparison functions in the configuration panel

Security Features for Tags

All tags that allow operators to write to a device can be secured with an application privilege.



This can be implemented as follows:
One of the tag's parameters will be named SecurityBit, as follows:

```
SecurityBit    <:TagField("SQL_VARCHAR(255)", "Security Bit",
10):>
```

The refresh module ensures that this contains the correct data type:

```
SecurityBit = Cast(SecurityBit, 1 { Short });
```

And, the configuration panel uses the PSecBit function to display the drop-down list and populate it with the existing application privileges.

```
{***** Security bit *****}
GUITransform(30, 180 { btm at 125 }, 470, 80,
      1, 1, 1, 1, 1            { No scaling                      },
      0, 0, 1, 0               { No movement; visible; reserved },
      0, 0, 0                  { Not selectable                  },
      \DialogLibrary\PSecBit(\#SecurityBit, \SecurityBitLabel, 1 { ID
}));
```

Each output widget will test the SecurityBit of the attached tag against the logged-in operator's privileges before performing a write. If you have created your own output widgets for your custom tags, you should also check the security privilege. An example of the code to do so follows:

```
   IfThen ( !Editing && \Code\OkToWrite(\SecurityBit, 1),
```

The value of \SecurityBit comes from the tag instance attached to the widget. The call to \Code\OkayToWrite() compares the logged in user's assigned security privileges with the SecurityBit and returns TRUE or

FALSE. It is then the job of the widget to either allow the write to proceed.

Containers, Contributors and Site Tags

To allow for a variable number of parameters for a tag, the concept of containers and contributors is supported within VTScada. A "container" is a tag that holds a collection of other tags known as "contributors". A tag may be a container for multiple contributor types, and a tag may also contribute to several different containers.
A site tag a type of container. It will commonly have the internal variables, LatitudeValue and LongitudeValue, although these are not required. The most important detail is that the groups declaration have at least the following line. Note that Site tags do not require the use of AddContributer, DeleteContributer or GetContributers.

```
[ (GROUPS)
   Shared Container { We are a container tag };
]
```

A tag that is a container may also be a contributor at the same time (a container tag may accept values from its contributors, while in turn contributing its value to another container).
For example, each station in a given area might contribute its value to a container that calculates the value for that area, and each area container might contribute its value to a container that calculates the value for the province or region. These regional containers can also contribute their values to a container that calculates a national value.
The container uses a "handle" variable that lists the contributors for a given tag. There may be several handle variables defined, one for each kind of contributor possible. There is no naming restriction on handle variables, but both the contributor and container modules must know the name. A handle variable is exclusively manipulated by three VTScada functions:

- AddContributor
- DeleteContributor
- GetContributors

Related Information:

...Overview of the AddContributor Function

...Overview of the DeleteContributor Function

...Overview of the GetContributors Function

...Latitude and Longitude for Site Tags

Custom Filtering of the Sites List and Map

The Sites page shows a list of all configured site pages in your application. For each of these, sub-sites and I/O are also listed. If you have a mixture of sub-site types and want to control which ones are displayed, or if you want to exclude the I/O, there are some ways in which you can filter the contents of the list.
The list is built automatically. Developers can filter this list for their Context-tag based sites by adding the parameter, CustomSiteListFilterType and setting its value to the name of the type or type-group that they want to filter for.
If you are writing a site tag from scratch, you can build-in filters for the sites list and for the maps by adding subroutine modules to be used as call-backs.

CustomSiteListGetSubTags

This must return an array of tag names. If this module exists in your custom tag, then GetSessionContainerTags will call it instead of GetTagList. An example of such a module is provided:

```
<
{================== CustomSiteListGetSubTags =================}
{=============================================================}
CustomSiteListGetSubTags
[
  Tags;
]
CustomSiteListGetSubTags [
  If 1;
```

```
  [
    Tags = \GetTagList(Root\Name, Invalid, Invalid, Invalid, "MultiS-
mart");
    Return(Tags);
  ]
]
>
```

After adding this code, you may want to consider setting your tag's SiteListDisplay parameter. If your subroutine returns a list of I/O, you may want to set the SiteListDisplay to 1 to treat it as a site. if it returns a list of sub-sites or a mixture of sub-sites and I/O, you may want to set SiteListDisplay to 2 in order to treat it as a folder. If the SiteListDisplay value is not set explicitly, the default behavior may be other than what you prefer.

CustomSiteMapGetSubTags

This module will be similar in structure to CustomSiteListGetSubTags. Two parameters can be provided:
ShowAll is an optional Boolean.
NavigationPath is an optional array of the tag names that were followed (clicked upon by the operator) to arrive at the list being viewed. This parameter can be used if you want to change the filter based on the path taken through the site structure.

```
<
{================== CustomSiteMapGetSubTags ==================}
{=============================================================}
CustomSiteMapGetSubTags
(
  ShowAll;
  NavigationPath;
)
Main[
  { Your filtering code, returning an array of tag names. }
]
>
```

Overview of the AddContributor Function

The "AddContributor" function adds a contributor to a container.
"AddContributor" is called from the contributor.
The parameters for the "AddContributor" function:

HandleName	The name of the handle variable in the container module.
ArrayName	The name of the variable in the ContainerObj parameter that holds an array of values to which the contributor's value should be added. This parameter may be invalid if there is no such array in the container.
CountName	The name of the variable in the ContainerObj parameter that holds a count of the current number of this type of contributor. "CountName" may be invalid if no such variable exists in the ContainerObj. Not all contributors need to be counted. The CountIncrement determines the initial change in the count, and the contributor must maintain the count.
ContainerObj	The object value of the container tag module.
ContributorObj	The object value of the new contributor to add.
IndexAddress	The address of the variable holding the contributor index.
Value	The current value to set in the container's ArrayName array. This value may be invalid, and may be updated at any time by the contributor by scoping into the ArrayName in the container and setting the array element at the index that will be set in the variable pointed to by IndexAddress (see above).
CountIncrement	This value is added to the CountName variable in the container (see above). This value is usually a "0" or a "1", indicating whether or not the contributor is actively contributing its value now. The contributor increments or decrements the value of the CountName variable (see above) as the corresponding state of the contributor changes.

Related Functions:

... AddContributor

Overview of the DeleteContributor Function

The DeleteContributor function removes a contributor from a container. "DeleteContributor" is called from the contributor.
The parameters for the "DeleteContributor" function:

HandleName	The name of the handle variable in the container module.
ArrayName	The name of the variable in the ContainerObj parameter that holds an array of values from which the contributor's value should be deleted.

	This parameter may be invalid if there is no such array in the container.
CountName	The name of the variable in the ContainerObj parameter that holds a count of the current number of this type of contributor. "CountName" may be invalid if no such variable exists in the ContainerObj. Not all contributors need to be counted. The CountIncrement determines the initial change in the count, and the contributor must maintain the count.
ContainerObj	The object value of the container tag module.
ContributorObj	The object value of the contributor to remove.
CountIncrement	This value is subtracted from the CountName variable in the container. This value is usually a "0" or a "1", indicating whether or not the contributor is actively contributing its value now. The contributor decrements the value of the CountName variable (see above) as the corresponding state of the contributor changes.

Related Functions:

... DeleteContributor

Overview of the GetContributors Function

The "GetContributors" function returns a copy of an array of object values of contributors for a given container.
The parameters for the "GetContributors" function:

HandleName The name of the handle variable in the container module.

ContainerObj The object value of the container tag module.

Related Functions:

... GetContributors

Latitude and Longitude for Site Tags

If users will place your container tag (site tag) on a map, it will need to store latitude and longitude coordinates and have a way to refresh those values.

The parameter list must include Latitude and Longitude, to provide a means for users to locate the site on a map:

```
Latitude   <:TagField("SQL_LONGVARCHAR" ):> { The location of this
site };
Longitude  <:TagField("SQL_LONGVARCHAR" ):> { The location of this
site };
```

Matching constants must be created, as for all tag parameters.
In the tag's list of variables, you will create the following:

```
LatitudeValue  { Evaluated location of this site };
LongitudeValue { Evaluated location of this site };
```

The main state for the tag must provide a way to set those internal variables:

```
TagMain [
  { Set the Latitude and Longitude }
  LatitudeValue = (Valid(Scope(Root, "Latitude", TRUE)) ? \Ex-
pressionManager : Invalid)\ToValue(Scope(Root, "Latitude", TRUE));
  LongitudeValue = (Valid(Scope(Root, "Longitude", TRUE)) ? \Ex-
pressionManager : Invalid)\ToValue(Scope(Root, "Longitude", TRUE));
]
```

And finally, the Refresh module must provide a way to update the variables:

```
...
\ExpressionManager\SafeRefresh(&Latitude, Parms[#Latitude]);
\ExpressionManager\SafeRefresh(&Longitude, Parms[#Longitude]);
...
```

Custom Help Systems

If properly configured, the tags in your VTScada application can be associated with a custom help file topic that the end user can open in one of two ways:

- From the shortcut menu's "Help" option;
- From the "Help" button that appears between the "OK" and "Cancel" buttons on the tag's properties folder if the tag has had its Help Search Key property configured with a valid topic ID.

In order to properly associate custom help file topics with the tags in your application, it is necessary to do the following:

1. Save your custom help file in your application directory.
2. Specify the custom help file in your application's configuration "HelpFile" variable (e.g. "HelpFile = MyCustomHelp.hlp" or "HelpFile = MyCustomHelp.chm").
3. In the Help Search Key property (on the ID tab of the tag's properties folder), enter the numeric map number that is associated with the help topic you wish to open for this tag.

See also: Integrating Custom Help Files Into VTScada.

Internally, the VTScada help system is called using a module named, "HelpLaunch". The HelpLaunch module spawns the help file identified in the "HelpFile" variable in the application's configuration variables. The only parameter passed to the HelpLaunch subroutine is the tag instance (object value). This subroutine displays a warning dialog if the HelpFile program is already open.

The parameter passed to the help file identified in the configuration variables is the tag's "HelpKey," which is the map ID identifying the topic to display in the help file. If the "HelpFile" variable is not defined, the built-in call to the VTScada "Help" function is used, ensuring that only one copy of the help program is open.

The Help command should be built as follows when the HelpFile variable defines a custom help file:

```
HelpFile HelpKey
```

The "HelpLaunch" module may be replaced with a custom version if required. The tag's scope is searched first for the "HelpLaunch" module, then the \Code, and the default. This enables a plug-in module to define a help module for each tag type. To hook the help into a Navigator call so that the help file appears when the Help option is selected in the shortcut menu, set "HelpLaunch" as the name of the module to launch with Invalid as the scope of where the module gets launched. It is called from the "ConfigFolder" dialogs, as well as from user code.

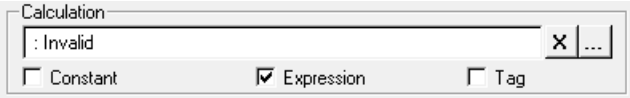
Related Information:

Additional information on related subjects is available in this guide. Please see:

- RPC Manager (see: RPC Manager).
- Adapting a Driver for VTScada (see: Communication Drivers).
- Alarm Manager (see: Alarm Manager Service).

Expressions as Tag Parameters

VTScada provides support to allow arbitrary expressions as tag inputs for some tag types and widgets. The PTypeToggle widget (following image), enables the user to choose between a tag, a constant value for a tag parameter, or a block of steady state VTScada code (i.e. an expression).



This chapter explains the modules that provide support for expressions and describes how to add a PTypeToggle widget to your custom tags. See also: Creating Expressions.

All tags and widgets in the VTScada layer that use a PTypeToggle have been modified to support expressions. Supported tags include:

- Function and Calculation tags (all types)
- Alarm tags
- Any tag with a built-in alarm (Analog Status, etc.)
- Logger tags
- MultiWrite tags

All tag widgets include expression support for Scaling, Movement and Visibility. Some, such as the Two Color Bar, Multi-color, Multi-text and Plot Data also support expressions in the Panel.

Related Information:

...ExpressionManager Usage for VTScada Programmers

...Adding Expression Support to an Application

...The ExpressionEdit Widget

...Issues and Risks

ExpressionManager Usage for VTScada Programmers

The modules used by the ExpressionManager, are as follows:

\ExpressionManager\Start(Script, ExpressionParent, ExpressionCaller [, ReturnErrors])

This method first attempts to cast Script to a number, then tries to scope it to a tag, then attempts to compile it into an expression module. If Script casts to a number or scopes to a tag, it is returned. If Script compiles successfully, it is returned. If Script compiles successfully, it is launched with ExpressionParent and ExpressionCaller as parent and caller respectively. If the cast, scope, and compile fail, it will return Invalid unless the optional parameter ReturnErrors is set to true, in which case a string describing the problem will be returned.
This interface is designed to behave in the same way as tag Refresh() methods do when interpreting strings that may be constants, tags, or expressions.

\ExpressionManager\IsExpression(ModuleRef)

This subroutine protects calling code from the design decision to mark ownership of expressions using a signature. It returns 1 if ModuleRef refers to a running module that the ExpressionManager launched, and 0 otherwise.

\ExpressionManager\ToString(Obj)

This subroutine will return a string representation of Obj. ToString() handles constants, tags, and expressions. \ExpressionManager\Start (ExpressionManager\ToString(Obj), …) will return a new instance of a running expression, or a reference to the same tag, or the same constant.

\ExpressionManager\SafeAssign(PTarget, Source)

This convenience routine checks to see if PTarget is a pointer to a running expression. If it is, the expression is slain. Then *PTarget is set to Source (*PTarget is always set, regardless as to whether or not it points to an expression).

\ExpressionManager\SafeCopy(Obj, ExpressionParent, ExpressionCaller)

The same as \ExpressionManager\Start(\ExpressionManager\ToString (Obj), ExpressionParent, ExpressionCaller).

\ExpressionManager\SafeRefresh(PRefreshedVar, ParmVar[, NewValue])

```
  PRefreshedVar{ Variable that is being refreshed.     };
  ParmVar{ Previous value of variable being refreshed. };
  NewValue{ Optional initial value for expression       };
```

If a tag parameter holds a running expression rather than a variable, care must be taken in refreshing it. The SafeRefresh method provides a convenient way for developers to ensure that all parameters are correctly refreshed, and at the same time ensures that similar operations use the same code path.

Using the Function tag as an example, its refresh module starts as follows:

```
Refresh [
  If watch(1);
  [
    {*************************************************}
    { Parameters to the Function                      }
    {*************************************************}
   {***** Update variables *****}
    \ExpressionManager\SafeRefresh(&P1, Parm[#P1]);
    \ExpressionManager\SafeRefresh(&P2, Parm[#P2]);
    \ExpressionManager\SafeRefresh(&P3, Parm[#P3]);
    \ExpressionManager\SafeRefresh(&P4, Parm[#P4]);
```

Here, the variables P1 through P4 are all parameters to the tag. SafeRefresh will instantiate the expression if necessary and will set the parameter to that expression. If the parameter is a tag reference or a simple constant, those will be handled correctly as well.

\ExpressionManager\ToValue(Parm)

Takes the return value from SafeRefresh and returns the current value of the expression, tag or constant. \ExpressionManager\ToValue should always be called in order to safely use the value of a parameter in a module.

Example:

The Function tag uses the
following to access the values of these four parameters:

```
RawValue = PickValid(Cast(\ExpressionManager\ToValue(P1), 3),
         Valid(P1) ? Invalid : 0) +
           PickValid(Cast(\ExpressionManager\ToValue(P2), 3),
         Valid(P2) ? Invalid : 0) +
           PickValid(Cast(\ExpressionManager\ToValue(P3), 3),
         Valid(P3) ? Invalid : 0) +
           PickValid(Cast(\ExpressionManager\ToValue(P4), 3),
         Valid(P4) ? Invalid : 0);
```

If the parameter is a constant or expression, the scope resolution references to the tag-Name and tag-Value variables would be undefined. Any addition of variables, such as child tags starting or another application starting would cause the code to re-trigger. This behavior is intentional in order to catch the addition of child tags AFTER the instantiation of the references to them.

Adding Expression Support to an Application

In the Refresh() method of the module, use SafeRefresh() to update any parameters that may be assigned an expression. The generic form is as follows:

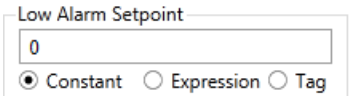
```
\ExpressionManager\SafeRefresh(&P1, Parm[#P1])
```

For example, the low alarm setpoint of an Analog Status tag can be defined by an expression. In the Analog Status tag's Refresh() module, the SafeRefresh statement for the low alarm setpoint is:

```
\ExpressionManager\SafeRefresh(&AlarmLo, Parm[#AlarmLo]);
```

Also, add a PTypeToggle to the config folder of the tag for each variable that may be an expression.

For example, here is the code that draws the user input for the low alarm setpoint of an Analog Status tag:




```
{***** Low Setpoint *****}
  GUITransform(30, 103, WIDTH/2 - 5, 45,
               1, 1, 1, 1, 1     { No scaling                      },
               0, 0, 1, 0        { No movement; visible; reserved },
               0, 0, 0           { Not selectable                  },
 \DialogLibrary\PTypeToggle(\#AlarmLo, "Numeric" { point type },
                            \LowAlarmSetpointLabel, 1 { to 3 - ID },
                            0 { top align }, 1 { align title },
                            Invalid, Invalid { limits },
                            Trigger, 1 { Allow Expression }));
```

To add expression support to a tag widget:
Often a tag widget variable X that can be a tag or a constant will be set in steady-state with PickValid(). Graphic calls then use X directly as an input. This won't work if X is extended to allow expressions, as Start() is a subroutine. Instead, introduce a new variable (for example, Xdata) and base the graphics on that. Then add new code similar to the following:

```
If Watch( 1, X );
[
\ExpressionManager\SafeAssign( &Xdata, ValueType(X) == 7 { Object } ?
X : ValueType(X) == 4 { Text } ? \ExpressionManager\Start(X, Self,
Self) : X );
]
```

Finally, make sure that there is a PTypeToggle in the Panel module for X, and ensure that the PTypeToggle is called with the EnableExpressions argument set to 1.

The ExpressionEdit Widget

The ExpressionEdit widget is used by the PTypeToggle to allow users to type in expressions. It can also be used on its own in cases where screen area is too limited for a PTypeToggle. It looks and acts similar to a SelectObject, except that instead of bringing up the Tag Browser when the user clicks the … button, an expression entry dialog is displayed. ExpressionEdit uses Start() to resolve what the user types in, so it can be used to select an expression, a tag, or a constant.
The generic form of the interface is as follows:

```
ExpressionEdit(X1, Y1, X2, Y2, ParmObjPtr, SetParm, ID, Root)
```

X1	Left or right side of graphic (*)
Y1	Bottom or top of graphic
X2	Right or left side of graphic
Y2	Top or bottom of graphic
ParmObjPtr	The parameter being set
SetParm	Set when the parameter has been set
ID	Focus ID
Root	Calling tag to which this will hook

Note: (*) Those not familiar with the VTScada graphic functions may find it confusing that X1 can be either the left or right of the graphic. Between X1 and X2, whichever is the smaller value is taken as the left and the larger value becomes the right. Similarly for Y1 and Y2.

Issues and Risks

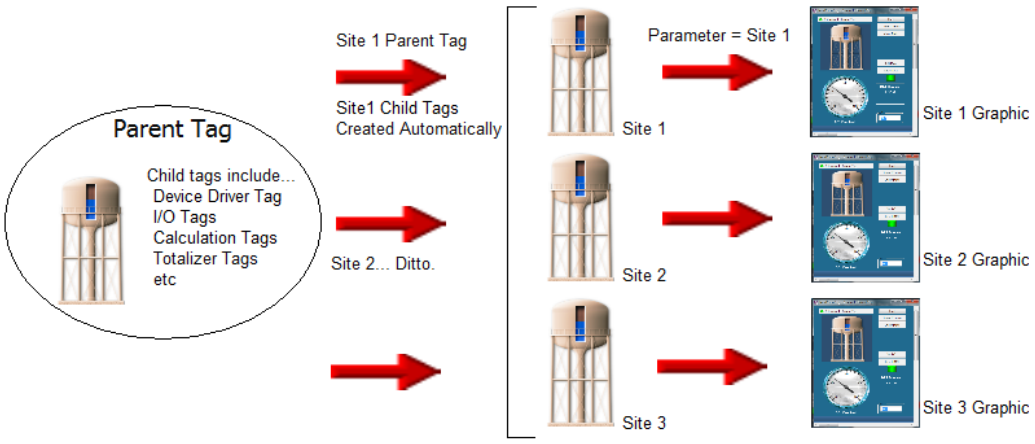
Care must be taken when assigning to and from variables that may hold expression values.
The implementation of expressions assumes that only one variable at a time holds a reference to a running expression.
A running expression must be slain before assigning to a variable that refers to it. Failing to do this will result in an orphaned expression, which is worse than creating a memory leak, because processor cycles are additionally wasted.
A reference to a running expression should not be copied. Instead of copying the reference, a new expression should be started with the same source. Failing to do this consistently will result in expressions dying unexpectedly.
The convenience methods \ExpressionManager\SafeAssign() and \ExpressionManager\SafeCopy() are provided to make this easier for developers.

Care must be taken not to rely on expressions running on different servers to return the same thing (example: Now(1))
It is possible, in theory, to create a situation where expressions are returning different values from each PC in a remote application. For example, suppose that a Logger is created that accepts input from a Calculation tag, which in turn uses an expression that relies on the local clock. If the local clock is slightly different on each PC, then this may result in odd log data when synchronizing between different PCs.

Programming Parent Tags

Note: The following information describes an older technology. With rare exceptions, Parent-Child tag structures, created using the Tag Browser, should be used instead of programmed parent tags.

Parent tags are a user-created tag type that will generate a set of related tags (child tags) with each new instance. They are typically used in applications that have a series of devices such as generators or lift stations where each instance will use a similar port, driver, I/O tag set, etc.
The tag that represents the generator or lift station as a whole is referred to as the "parent tag," while the various dynamically-generated tags are referred to as the "child tags". The child tags are created by way of calls to the StartTag function. Their parameters do not exist within the tag properties database, except in the case where the developer subsequently overrides those properties.
Since the child tags are created automatically by each parent, it becomes a simple task for developers to add new lift stations or other objects to the application.



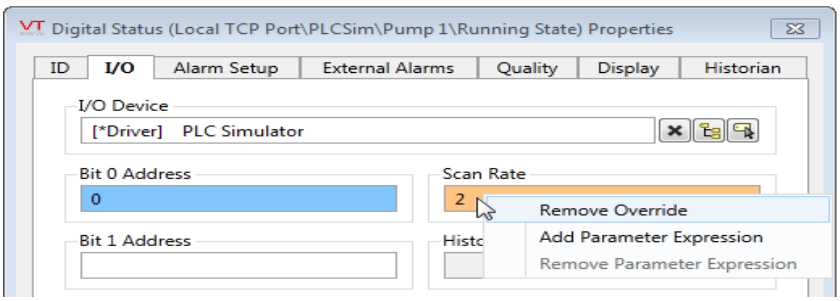
Parametrized pages can be created that accept an instance of a parent tag as the parameter, then automatically link all of the monitoring and control features within the page to the child tags. Parent tag structures are most useful when there is a clear pattern to the child tag parameters from one instance to the next. It should be possible to calculate each child tag's parameters from its parent's configuration.

Child tags and their properties are not stored in the tag properties database. The exception to this rule is that overrides to properties are stored in the tag properties database. Parent tags and all of the child tags will be listed in the Tag Browser and their properties may be edited there.

Note: Although child tags do not store their parameters in the tag properties database, they do count towards the tag limit as set by your VTScada license.

Overrides to child tag values will remain in effect and be distributed to all stations running the application. Overridden fields in the configuration panel will turn orange instead of green (subject to the application property, ParmOverrideColor). You can also right-click on a child tag's configuration panel field and select "Add Override" from the pop-up menu.

Right-clicking and selecting "Remove Override" is the only way to return to the property value defined in code – simply deleting the value will set it to an override value of Invalid, rather than removing the override.



Related Information:

...Building Parent Tags

...Widgets for Parent Tags

...Optimizations and Considerations When Using Child Tags

Building Parent Tags

The structure of a parent tag is the same as any other custom tag (see Creating Custom Tag Types) with one exception: The parent tag contains a number of calls to StartTag, thereby creating child tags automatically with each new instance.

The parent tag's parameters should include all of the information that will be required by the various child tags. There must be some pattern to the child tag parameters from instance to the next for these structures to be useful. For example, if all of the child tags that perform I/O functions use addresses that can be calculated from a station number or a base address value, then the parent tag will require very little configuration. If each parameter of each child tag in each instance of the parent must be provided individually, then few benefits remain.

For parent tags with very complex configurations, you should consider creating a configuration wizard to help operators create each new instance. See The VTScada Wizard Engine. If the configuration is not overly complex, then a standard set of configuration panels should suffice.

The concepts of parent tag structure are illustrated in the following example. This shows the source code for a simple lift station with a port,

a Modbus Compatible Device , a polling driver. I/O addressing in this example takes advantage of the Modbus virtual I/O feature.

```
{====================== Parent Tag Example ======================}
{ An sample Parent tag that creates several child tags.    }
{================================================================}
{ The parameters section of the module contains all of this      }
{   tag's configuration parameters                               }
(
  Name            <:TagField("SQL_VARCHAR(64)",  "Name"        ):>
    { Name of this tag                                 };
  Area            <:TagField("SQL_VARCHAR(255)", "Area"        ):>
    { Group which this device belongs to               };
  Description     <:TagField("SQL_VARCHAR(255)", "Description" ):>
    { Description of device                            };
  Station  <:TagField("SQL_VARCHAR(255)", "Station"     ):> = 1
    { Modbus address of the PLC                        };
  IOBaseAddr      <:TagField("SQL_VARCHAR(255)", "IOBaseAddr"  ):>
    { I/O addressing offset for this instance          };
  Port            <:TagField("SQL_VARCHAR(255)", "Port"        ):>
    { Port tag used to communicate with the device  };
  ScanInterval    <:TagField("SQL_VARCHAR(255)", "ScanInterval" ):>
    { Polling interval                                 };
)
{ The variables section of the module contains the standard     }
{   variables, constants and module declarations for a tag      }
[
  Root                  { Instance of this module                 };
  Value(5)              { Value of this tag                       };
  Refresh      Module   { Standard module in all tags             };
  ConfigFolder Module   { Configuration folder                    };
  Common       Module   { right-click menu and tool tip           };

  [(GRAPHICS)
    Shared Number;  { the only widget that will be available }
  ]
     { Parameter Constants }
  CONSTANT #Name          = 0;
  CONSTANT #Area          = 1;
  CONSTANT #Description   = 2;
  CONSTANT #Station  = 3;
  CONSTANT #IOBaseAddr  = 4;
  CONSTANT #Port          = 5;
  CONSTANT #ScanInterval  = 6;

]

{ The initialization state of all tags will set Root to self and call
}
{   the Refresh module
}

Init [
  If 1 Main;
  [
    Root = Self;
```

```
    Refresh();
  ]
]

Main [
{ The value of the parent tag will be taken from the value of the   }
{  Polling Driver child-tag                                         }
  Value = Variable("PollDriver\Value");
]

<
{========================= Refresh ======================}
{ This subroutine is called on startup and whenever the     }
{ tag's parameters change                                   }
{============================================================}
Refresh
(
  Parm         { Array for parameters prior to their change };
)

Refresh [
  If 1;
  [
    { This section creates the instances of the child tags. Some
}
    { parameter values come directly from the parent (Area), some are
}
    { set in code and some are built here, based on information from
}
    { the parent's parameters.                                  }
    { Since child tags are always named  ParentName\ChildName, there
}
    { is no need here to create a unique name here for each child
}
    { instance. The drivers, for example, will be
Instance1\SiteDriver, }
    { Instance2\SiteDriver, etc.                                }

    { Create Driver tag }
    \StartTag(Root,
              Valid(Root\Name)    { Conditional launch expression },
              "ModiconDriver"      { Tag type },
                { Name and value pairs }
              "Name",              "SiteDriver" ,
              "Area",              Area,  { The child takes its area
from the parent }
              "Description",       Concat(Description, " Driver"),
              "Station",           Station, { from the parent con-
figuration }
              "Channel",           0,
              "PortTag",           Port,    { from the parent }
              "TimeLimit",         1,
              "Retries",           3,
              "Hold",              0,
              "KeyOffDelay",       0,
              "Options",           0,
              "TimeSquelch",       0.2,
```

```
              "RetryTime",         1,
              "Adapter",           -1,
              "TimeInc",           2,
              "MBPRetries",        2,
              "MBPDMPathCount",    8,
              "MBPPollRate",       0.025,
              "VPLCHoldingCoils",  9999,  { Virtual I/O is used for
this example }
              "VPLCInputCoils",    9999,
              "VPLCInputRegs",     9999,
              "VPLCHoldingRegs",   9999,
              "HelpKey",           Invalid);
    { Create Polling Driver tag }
    \StartTag(Root,
              Valid(Root\Name) { Conditional launch expression },
              "PollDriver"     { Tag type              },
              "Name",          "SitePoller" { Parameter name & value
pairs },
              "Area",          Area,
              "Description",   Concat(Description, " Poll Driver"),
              "DeviceTag",     Concat(Name",\SiteDriver"),
              "PollGroup",     "DEFAULT",
              "Sequence",      0,
              "Interval",      ScanInterval,
              "Offset",        0,
              "PollDisabled",  0,
              "HelpKey",       Invalid);
    { Create an Analog Status tag }
    \StartTag(Root,
              Valid(Root\Name)    { Conditional Launch expression },
              "AnalogStatus"      { Name of tag type },
              "Name",    "AS1",
              "Area",    Area,
              "Description",   "Analog Status child 1",
              "DeviceTag",   Concat(Name",\SitePoller"),
              "Address",   Cast(40000 + Cast(IOBaseAddr, 2), 4),
              "ScanRate",   1,
              "UnscaledMin",   0,
              "UnscaledMax",   4095,
              "ScaledMin",   0,
              "ScaledMax",   100,
              "Units",    "%",
              "AlarmLo",   Invalid,
              "AlarmHi",   Invalid,
              "PriorityLo",   Invalid,
              "PriorityHi",      Invalid,
              "InhibitLo",   Invalid,
              "InhibitHi",   Invalid,
              "AlarmSound",   Invalid,
              "ManualValue",   Invalid,
              "Threshold",   Invalid,
              "Questionable",    0,
              "Quality",   Invalid,
              "DisplayOrder",    Invalid,
              "HelpKey",   Invalid,
              "PopupLo",   Invalid,
              "PopupHi",   Invalid,
```

```
            "AlarmLoDeadband",  Invalid,
            "AlarmHiDeadband",  Invalid,
            "AlarmLoDelay",    Invalid,
            "AlarmHiDelay",    Invalid);
   { Create an Analog Control tag }
   \StartTag(Root,
            Valid(Root\Name) { Conditional Launch expression },
            "AnalogControl"  { Name of tag type },
            "Name",   "AC1",
            "Area",   Area,
            "Description",  "Analog Control child 1",
            "DeviceTag",  Concat(Name",\SitePoller"),
            "Address",  Cast(40000 + Cast(IOBaseAddr, 2), 4),
            "UnscaledMin",  0,
            "UnscaledMax",  4095,
            "ScaledMin",  0,
            "ScaledMax",  100,
            "Units",   "%",
            "SecurityBit",   Invalid,
            "DataSource",   Invalid,
            "Questionable",  0,
            "DisplayOrder",  Invalid,
            "HelpKey",  Invalid);
   { Create a Digital Status tag }
   \StartTag(Root,
            Valid(Root\Name) { Conditional Launch expression },
            "DigitalStatus" { Name of tag type },
            "Name",       "DS1" { Name and value pairs },
            "Area",         Area,
            "Description",  "Digital Status child 1",
            "DeviceTag",    Concat(Name",\SitePoller"),
            "Bit0Address",  Concat(Cast(40100 + Cast(IOBaseAddr,
2), 4), "/0"),
            "Bit1Address",  Invalid,
            "ScanRate",     Invalid,
            "InvertInput",  0,
            "OffText",      "Off",
            "OnText",       "On",
            "AlarmState",   Invalid,
            "AlarmDelay",   Invalid,
            "TripOptions",  0,
            "Priority",     0,
            "Inhibit",      1,
            "AlarmSound",   Invalid,
            "ManualValue",  Invalid,
            "Questionable", 0,
            "Quality",      Invalid,
            "DisplayOrder", 1,
            "HelpKey",      Invalid,
            "PopUp",      0);
   { Create a Digital Control tag }
   \StartTag(Root,
            Valid(Root\Name)    { Conditional Launch expression },
            "DigitalControl"    { Name of tag type },
            "Name",             "DC1" { Name and value pairs },
            "Area",             Area,
            "Description",      "Digital Control child 1",
```

```
              "DeviceTag",        Concat(Name",\SitePoller"),
              "Address",          Concat(Cast(40100 + Cast
(IOBaseAddr, 2), 4), "/0"),
              "InvertOutput",     0,
              "PulseDuration",    0,
              "FeedBackTag",      Invalid,
              "FeedBack1",        Invalid,
              "FeedBack0",        Invalid,
              "SecurityBit",      Invalid,
              "DataSourceTag",    Invalid,
              "InvertDataSource", 0,
              "Questionable",     0,
              "DisplayOrder",     Invalid,
              "HelpKey",          Invalid);
    Return(0);
  ]
]
{ End of Refresh }
>
 { As with all tags, the ConfigFolder module is called when the }
 { tag's  properties are displayed.                    }

<
{=================== ConfigFolder ==========================}
{ This is the shell of an editable ConfigFolder for a tag     }
{ where the ConfigFolder is created and maintained by the     }
{ ConfigFolder Wizard.                                        }
{ Manual editing of the code is allowed as long as the state  }
{ structure is maintained.                                    }
{============================================================}
ConfigFolder
(
  Parms              { Pointer to array of parameters             };
  Current            { Currently selected tab (starts at 0)       };
  PtrWaitClose       { Pointer to FLAG - TRUE when wait to close.
                       Caller must default to 0.          };
  OKPressed          { OKPressed from PropertiesDialog            };
)
[
  Constant #WIDTH = 500;
  [(1)
    IDTabLabel      = "ID";
    ConfigTabLabel  = "Config";
  ]
]
Switch [
  If Current == 0 ID;
  If Current == 1 Config;
]
ID [
  {**** If page changes, change states *****}|
  If Current != 0 Switch;
  {***** Name of the point *****}
  GUITransform(30, 90, 470, 45,
               1, 1, 1, 1, 1           { No scaling
},
               0, 0, 1, 0              { No movement; visible;
```

```
reserved },
               0, 0, 0                  { Not selectable
},
              \DialogLibrary\PEditName());

  {***** Group (area) that the point belongs to *****}
  GUITransform(30, 200 { btm at 145 }, 470, 100,
               1, 1, 1, 1, 1            { No scaling
},
               0, 0, 1, 0               { No movement; visible;
reserved },
               0, 0, 0                  { Not selectable
},
               \DialogLibrary\PAreaSelect(1 {can edit}, 2 {ID}));
  {***** Description of the point *****}
  GUITransform(30, 200, 470, 155,
               1, 1, 1, 1, 1            { No scaling
},
               0, 0, 1, 0               { No movement; visible;
reserved },
               0, 0, 0                  { Not selectable
},
               \DialogLibrary\PEditField(2, \DescriptionLabel, 4
{text},
                                         3 {ID}));
]

Config [
    {**** If page changes, change states *****}
  If Current != 1 Switch;
    {***** Communications Port *****}
  GUITransform(30, 90, #Width - 30, 45,
               1, 1, 1, 1, 1,
               0, 0, 1, 0,
               0, 0, 0,
               \DialogLibrary\PSelectObject(#Port    { Parm
},
                                            "Ports"  { Point type
},
                                            "Communications Port" {
Title },
                                            1        { Focus ID
},
                                            0        { List at top
},
                                            0        { Align top of
bevel )));
    {***** Station Address *****}
  GUITransform(30, 145, #Width / 2 - 10, 100,
               1, 1, 1, 1, 1            { No scaling
},
               0, 0, 1, 0               { No movement; visible;
reserved },
               0, 0, 0                  { Not selectable
},
               \DialogLibrary\PEditField(#Station  { Parm Num
```

```
},
                                        "Station Address" { Title
},
                                        1               { Short
},
                                        2               { Focus ID
}));
    {***** I/O Base Address *****}
  GUITransform(30, 200, #Width / 2 - 10, 155,
               1, 1, 1, 1, 1            { No scaling
},
               0, 0, 1, 0               { No movement; visible;
reserved },
               0, 0, 0                  { Not selectable
},
               \DialogLibrary\PEditField(#IOBaseAddr  { Parm Num
},
                                        "I/O Base Address" { Title
},
                                        1               { Short
},
                                        2               { Focus ID
}));
    {***** Scan Interval *****}
  GUITransform(#Width / 2 + 10, 145, #Width - 30, 100,
               1, 1, 1, 1, 1            { No scaling
},
               0, 0, 1, 0               { No movement; visible;
reserved },
               0, 0, 0                  { Not selectable
},
               \DialogLibrary\PEditField(#ScanInterval { Parm Num
},
                                        "Scan Interval" { Title
},
                                        3               { Float
},
                                        3               { Focus ID
}));
]
{ End of ConfigFolder }
>

<
{======================== Common ============================}
{ This module handles the common actions associated with all    }
{ drawing modules for this point. It will be called by all     }
{ external drawing modules.                                     }
{==================================================================}
Common
(
  Left          { Area occupied by the drawing object         };
  Bottom;
  Right;
  Top;
)
```

```
Common [
  \PostIt(Left, Bottom, Right, Top, \Name, \Description,
          ! \GetUserSession()\NavActive);
  \Navigator(1 { Opening condition for the folder },
             Left, Bottom, Right, Top    { Target area for opening -
                                           same as the GUI statement
                                           area.    },
             { Menu line 1 } \PropertiesLabel, Invalid, 0, Invalid);
]
{ End of Common }
>
```

Widgets for Parent Tags

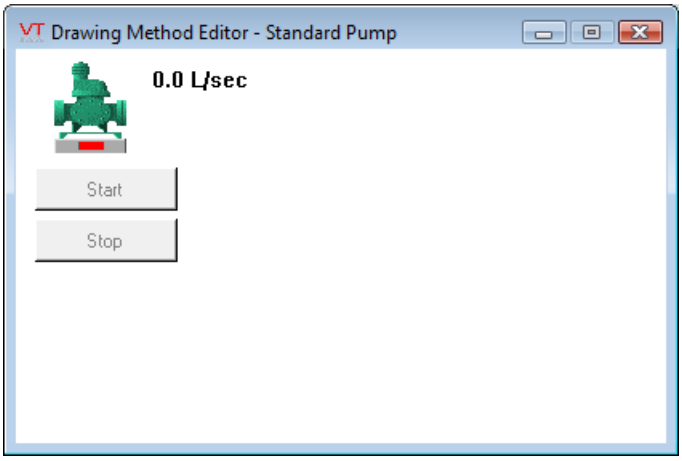
You can create a Draw method for a parent tag in the same way that you would for any other tag (see: Drawing Tags). A simpler method is available, however, which is to configure a parametrized page or a User Draw object.

Parametrized pages and user-created widgets are particularly well suited for drawing your stations and are an efficient alternative to writing graphics code in the template.

The process for adapting a parametrized page or widget to display all the child tags for a parent tag instance is as follows:

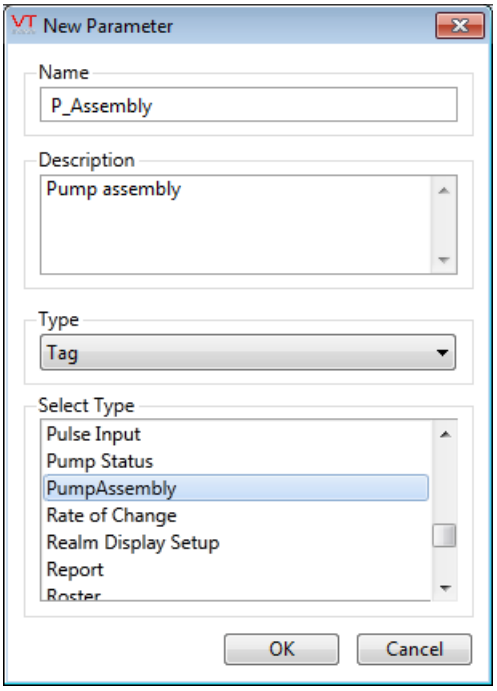
Given a parent tag that has been added to an application and where at least one instance of the tag has been created.

1. Create either a new parametrized page or a user widget.
2. Add a parameter to the parametrized page or widget. This will be a tag type where the selected type is the parent tag structure.

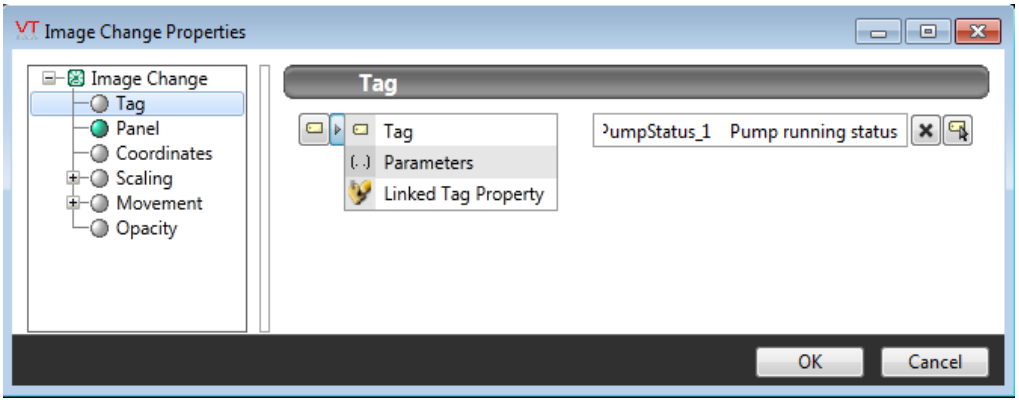


3. Open the page for editing.
4. Select an instance of the parent tag for the parameter.

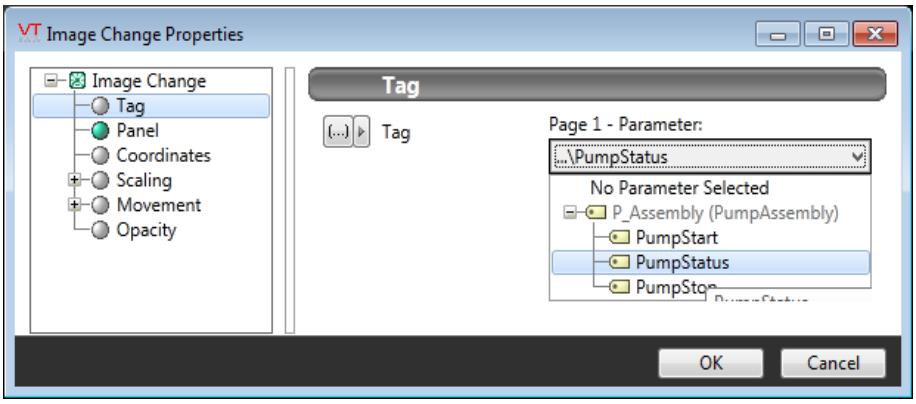
5. Draw the station using the child tags.



6. For each widget that makes up the station, change its source from tag to parameter.



7. Select the parameter, then the child tag or grand-child tag from the parent tag.



8. Save your work.

Optimizations and Considerations When Using Child Tags

The following optimizations and considerations should be heeded when you are using parent and child tags in your application:

- When structuring a parent and child tags, it is good practice to configure the parent with only those parameters that are necessary to define what is unique for the station (i.e. the parent should have enough parameters to cover variants in the children). The child tags will then have the common parameters defined.
- Don't launch tags that you do not need. If stations don't use all, launch only the ones required. For example, some stations may have 3 pumps, but others only 2.
- Child tags do count towards the tag limit as defined by your VTScada license.

Debugging and Analysis

VTScada provides you with a set of utilities to assist with debugging and analysis. These tools can help take the guesswork out of solving problems that might occur while you are developing your applications.

Related Information:

...Coordinates Application - enables you to precisely determine the coordinates of the mouse pointer.

...Debugger Utility - used to examine the contents of modules and their properties.

...Instance Count Application - provides a count of the instances (or copies) of modules that are running.

...Memory Tracer Application - used to analyze VTS's demand on computer memory.

...Profiler Application - analyze your applications to discover statements that are placing an excessive load on system resources.

...Source Debugger - the most powerful and comprehensive of the VTScada debugging tools.

...Test Framework Application - enables VTScada programmers to test VTScada applications.

...Thread List Application - provides a list of the separate threads of execution for which VTScada is responsible.

...Trace Viewer Application - monitor the content and parameters of driver messages and VTS-related network traffic as it occurs.

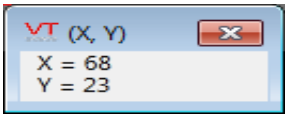
...Trace VTScada Actions Application - select VTScada services and actions and monitor by saving pertinent data to disk.

Coordinates Application

Included with VTScada is a simple, yet useful utility application named "Coordinates". This utility is useful when you need to determine the horizontal and vertical coordinates of the mouse pointer within any VTScada window (page or dialog).

Note: If the Coordinates application is not referenced in the VAM, you must manually add it. The Coordinates application's directory is named "XY", and is stored within the VTScada installation directory.

When you run the Coordinates utility application, it appears as shown:



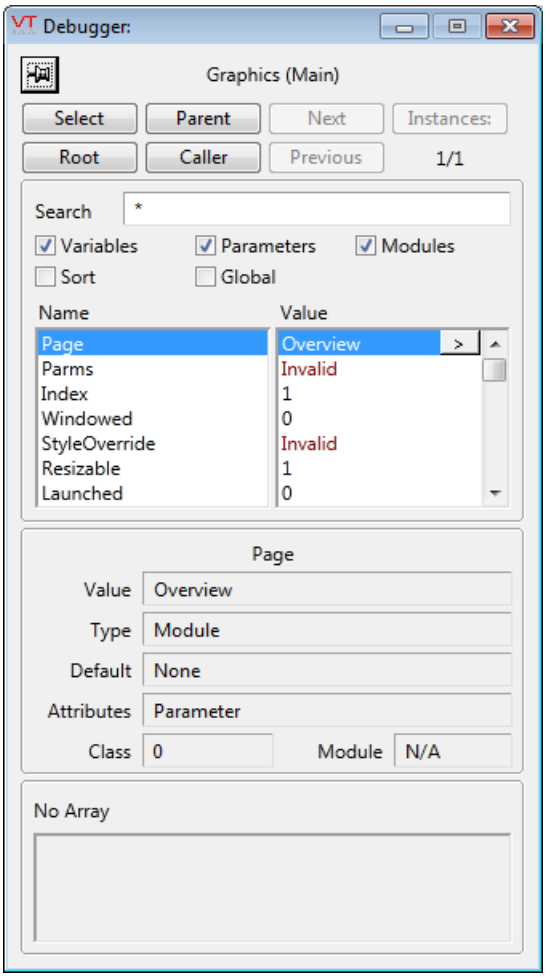
To use the Coordinates utility application to determine the X and Y coordinates of a page or dialog, run your application, navigate to the page or dialog whose coordinates you require, then position your mouse at the place you are interested in. The X and Y coordinates of the mouse pointer's position are displayed in the Coordinates application window.

Debugger Utility

The Debugger utility can be used to examine the contents of modules and their properties and can be invaluable in helping you to locate problems that happen only while your application is running. The information displayed in the Debugger is updated in real-time as the scripting code runs. This is an older technology than the Source Debugger, but it is still useful.

Note: If the Debugger application is not referenced in the VAM, you must manually add it. The Debugger application's directory is named "Debugger".
Pressing the keys "Ctrl-D" in the VAM will launch the Debugger.

An example of the Debugger:



The top of the Debugger displays the name of the module being examined and the name of the current active state, in parentheses following the module name.

Pin button

 (pin / currently pinned)

Use this button to choose whether the Debugger window should remain on top of all other windows.

Select button

Use this to choose which graphics module to examine.

1. Click the Select button.
2. Click on a VTScada page to begin examining the variables in the graphics module for that page.

Parent button

After selecting a graphics module, use the Parent button to examine the parent of the current module.

Next button and Previous button

Step forward and backward through the history of modules you've recently viewed.

Instances button

The Instances button enables you to step through the different instances of the module if more than one is running. The current instance and the total number of instances are shown beneath the button.

Root button

Used to quickly select the root module of the current module.
For example, if you are interested in examining a tag that is drawn on a page, you would first use the Select button to choose the page, then the Root button to move up the module tree. The display would then include (among other things) all of the tags in the application, from which you could select the tag you are interested in.

Caller button

Use this to examine the caller of the current module.

Search field

Use by entering a search string by which you wish the list of variables to be restricted. The wildcard characters '?' and '*' may be used in combination with either a single character or 0 or more characters to locate specific variables. You can use the asterisk wild card to stand for any combination of characters. For example, L* will return all variable names that begin with the letter "L"; *L or *L* will return all variable names that contain an "L" anywhere in the body of their name.
Pressing the key combination, "Alt-Home" will perform the same function as clicking on the Root button.

Variables check box
Filters the list to include variables when selected.

Parameters check box
Filters the list to include parameters when selected.

Modules check box
Filters the list to include modules when selected.

Sort check box
Filters the list alphabetically when selected. If deselected, the items displayed in the list will appear in the order in which they were declared.

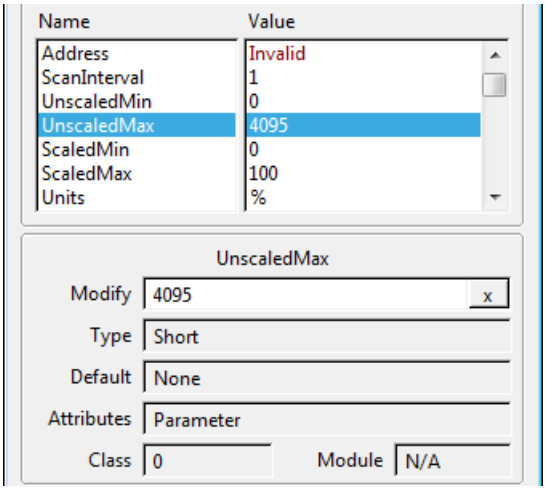
Global check box
Filters the list to include all variables that can be accessed from the current module; otherwise, only the variables in the current module will be displayed.

Note: When a new value is entered in the edit fields in the debugger, the Enter key must be pressed or it will not "register" (i.e. changing focus does not automatically enter values into the fields of the debugger). Further, if a text string is greater than 255 characters, only the first 255 characters are shown in the value field, but its full length is still registered.

Beneath the filtering and sorting check boxes appears the list of modules, variables and parameters. This list may be resized by dragging the divider between the Name and Value columns. If an item in the list is a module, an expand module button ">" will appear to the right of its "Value" column.

By clicking on this button you can expand the module and examine its contents.

Under the list of modules, variables and parameters is information about the items in the list. (example shows the list for an Analog Input tag with the UnscaledMax parameter selected)



Clicking on any item in the list of modules, variables and parameters will bring up information on the selected item, such as the value of the selected item, the data type (if the selected item is a variable), the attributes of the selected item, the class of the item, and if the selected item is a variable and the "Global" check box is selected, the module to which the variable belongs will be displayed. The name of the selected module, variable or parameter is centered above the Value field.

In the event that a selected variable has a value that can be modified, the Value field becomes enabled for modification, and includes a "clear value" button, which is marked with an "X". Any changes you make to the value of a selected variable will have an immediate effect in the running instance of the current module. Clicking on the clear value button will invalidate the current value.

If the selected variable is an array, the information displayed applies to the current element of the array only. The array contents are visible in the Array field at the bottom of the Debugger.

In the array display area, there is a scrollable listbox that contains a list of elements for the selected dimension. The current dimension can be changed at the top of this area. Selecting an element in this listbox will

allow information on the element to be displayed/edited in the variable information area.

Related Information:

...Source Debugger

Instance Count Application

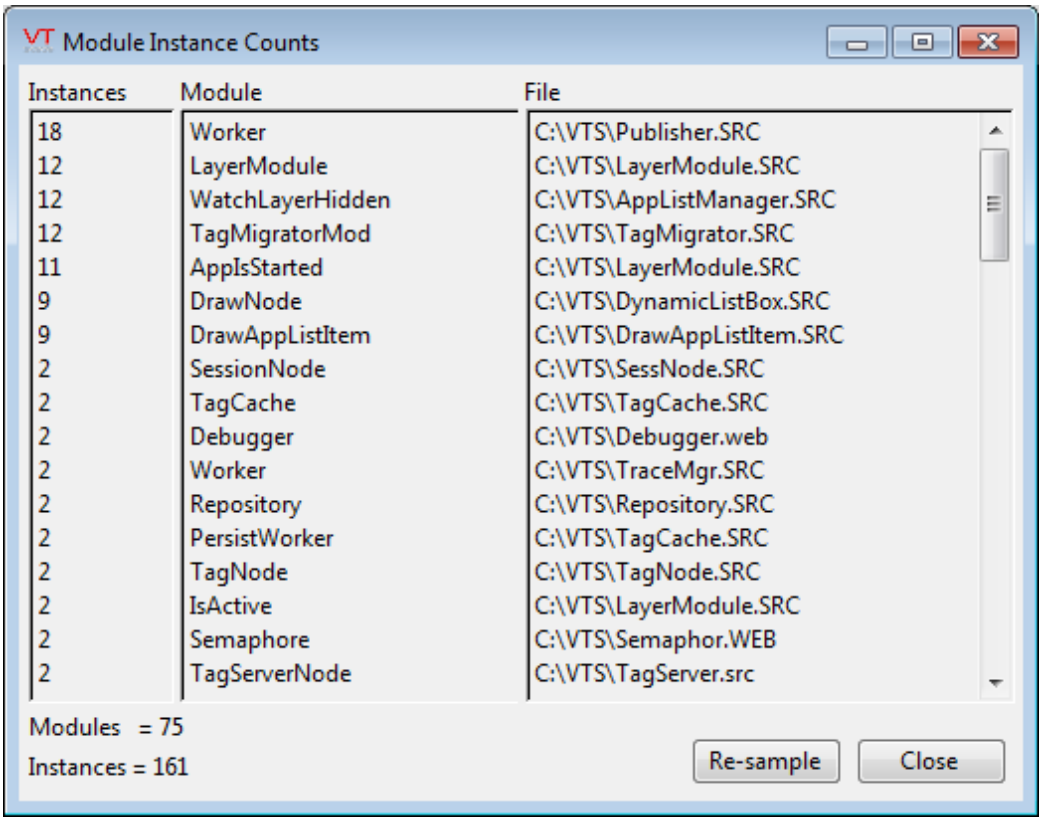
The Instances utility provides a count of the instances (or copies) of modules that are running in all VTScada applications on the workstation.

Note: If the Instances application is not listed in the VAM, you must manually add it. The Instances application's directory is named "Instance" and is stored within the VTScada installation directory.

The Instance tool can help you find potential problems in your application code. For instance, a high count for a particular module might indicate either:

- Excessive numbers of instances of a module have been launched.
- Module instances fail to stop executing when they should.

An example of the Instances utility:

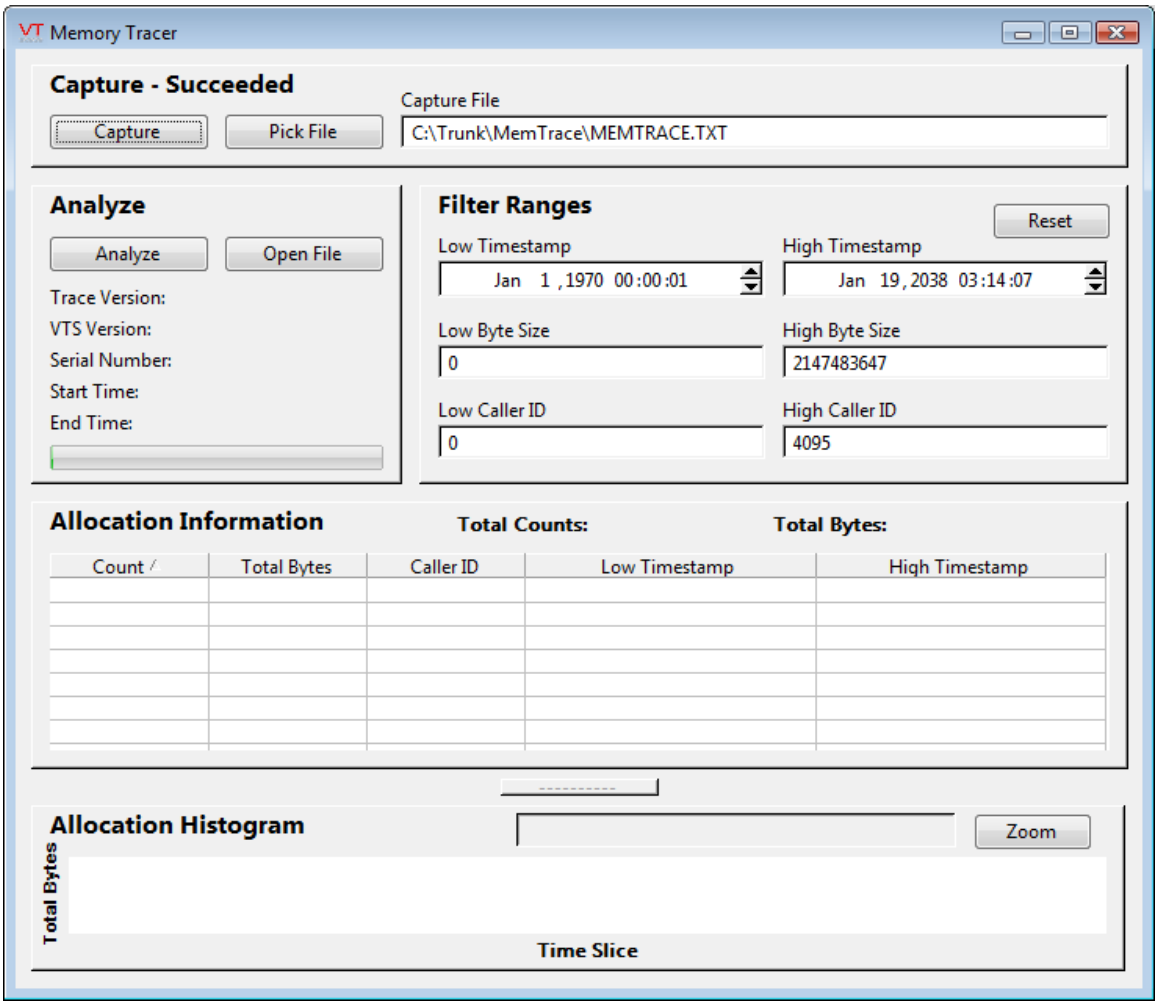


As shown, the Instances utility provides a count of how many instances of each module are running, and the file name where the module can be found. (Most of the files listed are accessible only to Trihedral employees.)

At the bottom left of the dialog are two values showing the total number of modules running and the total number of instances of those modules. The display is not updated automatically. You can use the Re-Sample button, found at the lower right of the dialog, whenever you want a current count.

Memory Tracer Application

The Memory Tracer application can be used by Trihedral staff to analyze VTS's demand on computer memory. It is often used to identify potential memory leaks when developing new engine code. The Memory Tracer application can be launched from the VAM.



Related Information:

...Using the Memory Tracer Utility – Instructions

...Analyzing a Memory Trace File – Description of the information captured

...Sorting Data in the Allocation Information List – Find and use the sorting features

...Viewing Smaller Segments of a Time Slice – Find and use the display features.

Using the Memory Tracer Utility

VTScada memory allocation calls are each given a unique ID. A memory leak or other problem would be indicated by an unusually high count or

total byte value, or by an allocation histogram that is not mostly empty. By referencing the caller ID associated with a high count value or that shows up many times during the time slice, programmer's can quickly find the code that is responsible for allocating this memory.
To begin capturing the VTScada memory usage, click the Capture button. The button will read "Capturing" for several seconds while data is being saved to the specified text file.
By default, VTScada creates a default memory trace file in the format of an encrypted text file named, "MemTrace.txt," which is stored within the MemTrace subdirectory of the VTScada installation directory (e.g. C:\VTScada\MemTrace\MemTrace.txt)
You may use the Pick File button or the Capture File field to specify an alternate location or file name if you wish. Memory Trace files may be used to send a trace to another computer for analysis.
The Filter Ranges, are used to restrict the capture to:

- VTScada usage during a given time frame
- Low and high byte sizes
- Low and high caller ID values.

The Reset button in this area will restore the filter to the widest possible ranges.

Analyzing a Memory Trace File

Once you've captured data to a memory trace file, or opened an existing file, you can analyze its data. To do so, simply click the Analyze button. When the progress bar reaches 100%, the Memory Tracer application's elements will be populated with data.
For each Caller ID (identifying the line of VTScada code that allocated the memory) the following information will be displayed:

- The count of times that line was called.
- The total memory allocated by that code
- The Caller ID
- Timestamps, identifying the earliest and latest times that code identified by this ID allocated memory during the capture timeframe.

The histogram will normally display a large number of total bytes at the beginning of the time frame, with a few small byte allocations during the span. As you move the mouse across the graph, the date and time matching the cursor location will be displayed. Pausing the mouse over any blip in the histogram will cause a window to be displayed showing the bytes allocated at that point in time, and the Caller ID's associated with code that allocated memory at that time.
Note that you can copy information from the Allocation Information table with a Control-C. This information can be pasted into any text file or spreadsheet. The copied information will not be sorted.

Sorting Data in the Allocation Information List

You can sort data in the Allocation Information list using the column headings. Click the column heading by which you wish to sort, and an arrow icon will appear to the right of the column heading. An upwards-pointing arrow indicates that the column is being filtered from low to high values, while a downwards-pointing arrow indicates that the column is being filtered from high to low values.

Viewing Smaller Segments of a Time Slice

The Memory Tracer application is used to zoom in on data displayed in the Allocation Histogram.
Position the mouse pointer at the starting of the area you wish to zoom in on.
Click and drag to select the entire area you wish to zoom in on. The selected area will be highlighted in grey.
Click the Zoom button. The Analyze process will repeat for just the selected time span and the Allocation Information list will display only those records that are associated with the selected zoom area.

Profiler Application

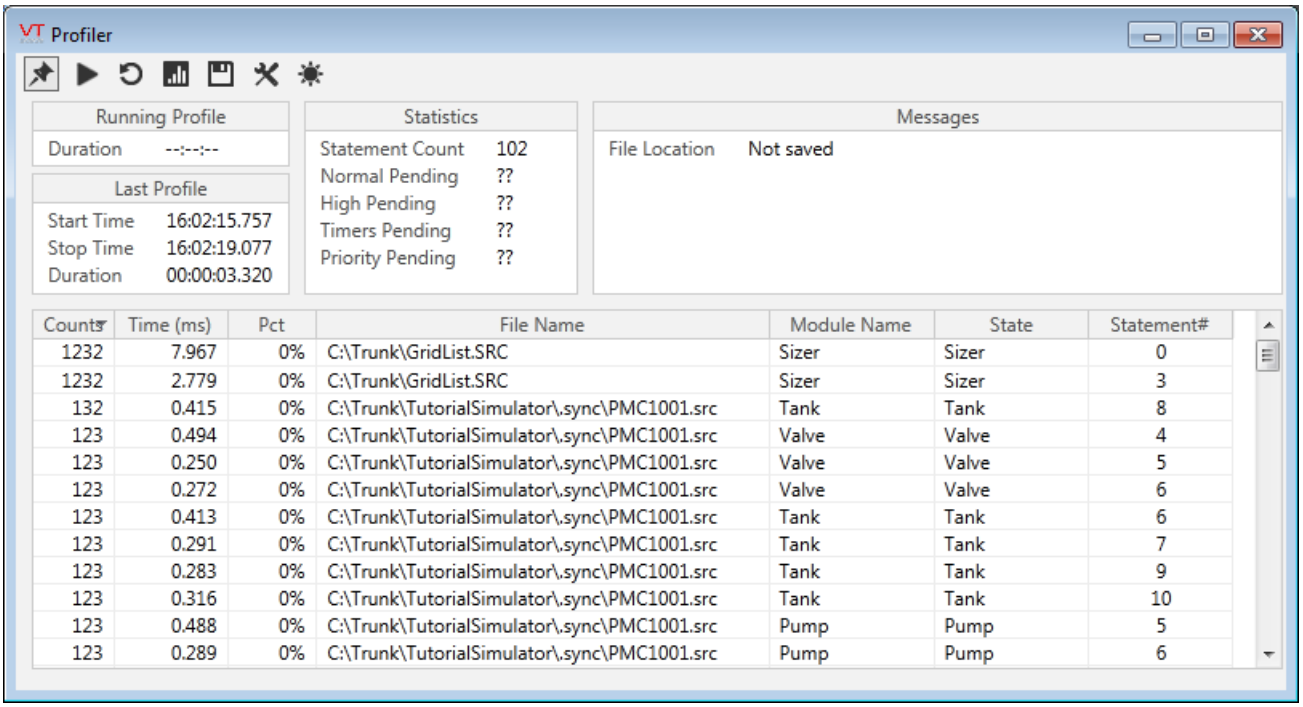
The Profiler can be used to analyze your applications to discover statements that are placing an excessive load on system resources. If a particular statement runs for a large percentage of the data collection time, it may be due to an "If 1" (continuously executing) condition. A statement that is called an extremely large number of times may indicate an inefficient algorithm in your code. Statements of this nature can have a severe impact on the speed and performance of an application. The Profiler can help you recognize such statements and correct any problems.

Note: The meaning of "high number" will vary depending on your application. It is helpful to run the profiler at different times and under different conditions to gain a sense of what is normal for any given application.

The Profiler analysis tool can be used in two ways: by collecting statistics about running statements during a defined collection period, or by taking an instant count of the number of statements of various priorities that are pending at the moment the Stats button is clicked.

An example of the Profiler follows. Note the "Settings" button. This will open a configuration dialog that can adjust how the profiler works. See: Profiler Settings. The notes in the Profiler Settings topic contain important information for anyone who intends to save the data collected by the profiler.

If the Profiler application is not referenced in the VAM, you must manually add it. The Profiler application's directory is named "Profile" and is stored within the VTScada installation directory.



The Profiler utility includes the following elements:

Command Buttons

Pin – When selected, the Profiler will always be the top-most window.

Start/Stop – Begins data collection. While the profile is running, the start button will become a Stop button.

Clear – Clears the history of the last profile collected. Does not apply to display of pending statements, collected by the Stats button.

Stats – Click to collect information about the number of pending statements of various priorities. (These are described in the list of displayed information, later in these notes.)

Save – Saves the data collected in the last profile to a CSV text file. This file will be saved to the C:\VTScada\Profile\Data folder and will have the name Year-Month-Day.txt. (2011-07-15.txt)

> Note: If no file is created, check the thresholds in the Settings dialog – it may be that no statement meets the minimum thresholds as configured and therefore, there is no data to save.

Settings... Opens a dialog that provides configuration options for the profiler. Described in the following topic.

Displayed Information

Last Profile – Shows the start time, end time and duration of the last data collection period.
Running Profile – Displays an elapsed time count in seconds while a profile is being collected.
Statement Count – Shows the number of statements that executed during the most recent profile.
Normal Pending – Shows the number of normal statements that were awaiting execution when the Stats button was clicked. For example, if a tag value changed just before you clicked the stats button, then there may be statements pending execution to react to that tag value change. If the Normal Pending value is very high, it may indicate that the script-execution engine is heavily loaded. This value cannot be taken by itself to indicate system loading since an IF-1 condition may create a large load while only adding 1 statement to the list of those pending.
High Pending – Similar to Normal Pending, but shows the number of high priority statements that were awaiting execution when the Stats button was clicked.
Timers Pending – Shows the number of time-related functions that were pending when the Stats button was clicked. A large number of these may indicate poor design.
Priority Pending – Shows the number of priority statements that were awaiting execution when the Stats button was clicked. Steady-state priority functions generally place a heavy load on system resources.

Grid Display Items

Counts –The number of times each statement executed during the most recent profile.
Time (ms) – The execution time (in milliseconds) for each statement in the most recent profile.
Pct – The percentage of time that each statement executed during the most recent profile.
File Name – The full path to and name of the source file containing each statement.

Module Name – The name of the module (within the identified source file) that contains each statement.
State – The state (within the identified module) that contains each statement.
Statement# – The numerical index of the statement within the identified state.

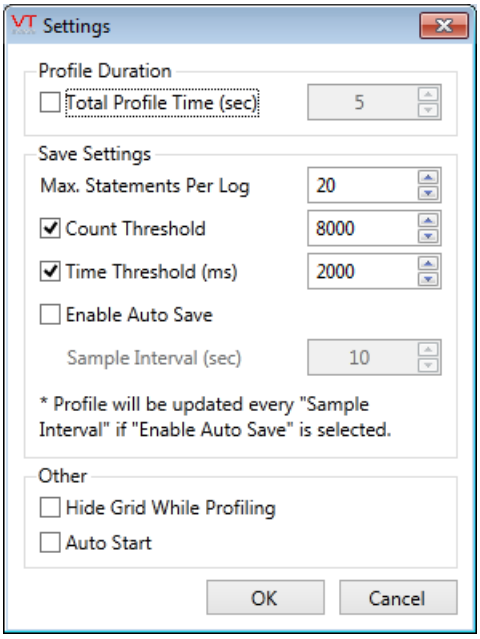
Note: When counting statements in a state, the Profiler begins at "0" and counts all steady-state statements from the top of the state to the bottom. The Profiler then returns to the top of the state and begins counting the statements within the first script, then the statements within the second script, and so forth.

Related Information:

...Profiler Settings

Profiler Settings

This dialog is used to control how the profile collects information.



Profile Duration

You may set a fixed length of time for the data collection period. This value is always in seconds – if you enter a decimal value it will be

rounded down to the nearest full second. If a duration is set, you can still use the Stop button to end a data collection period whenever you wish.

Save Settings

Options in this box control how the profiler saves information to a file. The file is saved to the Data folder under the Profiler application directory and will have the name Year-Month-Day.txt. One file will be created per day – multiple saves to file within a day are appended to the end of the day's data file.

Max. Statements Per Log

The data file will contain information sorted in decreasing order by count, by running time or both. Within each section, there will be Max Statements Per Log rows.

Count Threshold

When checked, the file will contain statements sorted by execution count in decreasing order. At least one statement must be executed this many times within the data collection period to be included in the saved file. Assuming that one statement crossed the threshold level, exactly Max Statements Per Log rows will be included, regardless of how many statements crossed the threshold level.

Time Threshold (ms)

Same as Count Threshold, except that this group of statements is sorted by total execution time during the collection period.
If neither County Threshold or Time Threshold are selected, the profile results for logging are sorted by whichever column, and in whichever order is selected in the grid list display.

Enable Auto Save

When checked, a fresh save will occur each Sample Interval seconds while the profiler is collecting data.

Hide Grid While Profiling

The statements that create the grid display in the profiler can be executed many times while data is being collected and will be included in the statistics. You may choose to disable the grid during data collection.

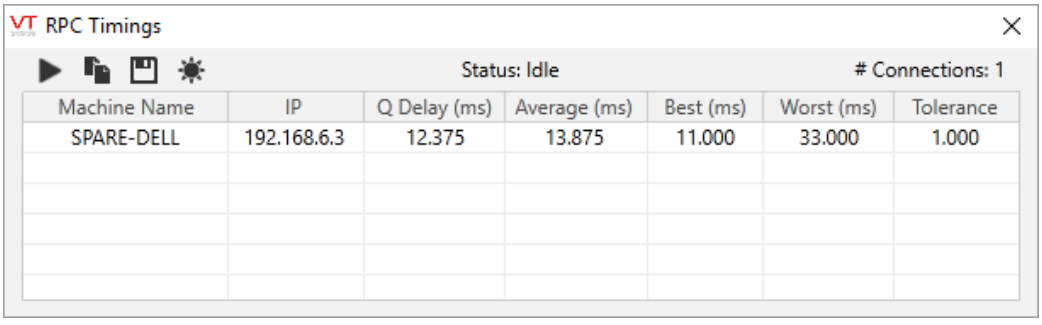
Auto Start

When checked, the profiler will begin collecting data as soon as it starts. You may use a set collection time, or the stop button to end the collection time period.

RPC Timing Utility

The RPC Timing Utility is included with VTScada, but not added to the default list of applications in the VAM. Use the Find Existing option of the Add Application Wizard to add it to your VAM.

This utility provides an assessment of remote procedure call round-trips between the servers running an application. Use it to identify timing issues, especially in busy systems. It can also identify servers that are using unintended IP addresses (non-SCADA network cards) and servers that are being overloaded with RPC calls.

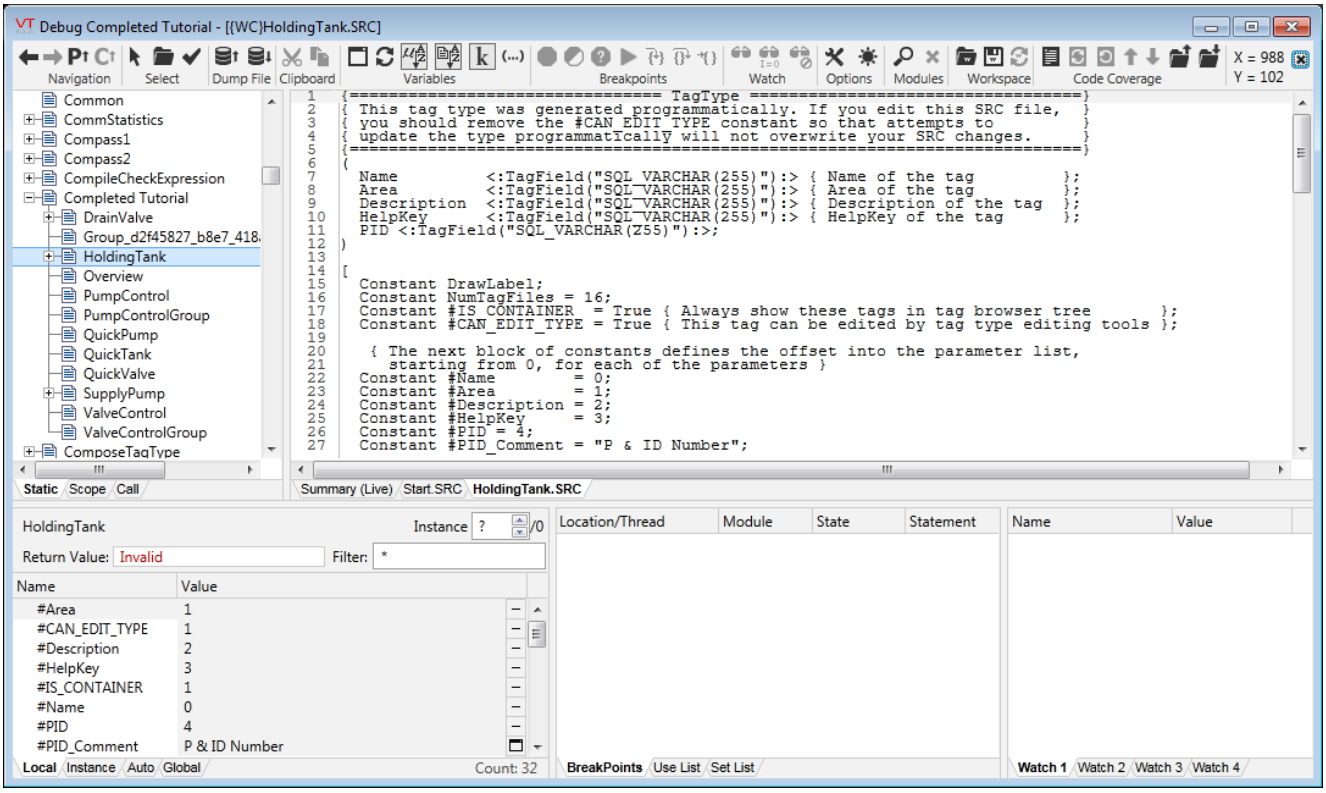


Timings are relative to the workstation where you run this application (never shown in the list). To refresh the display, click on the run button. A fixed number of test messages will be sent, from which timing statistics will be gathered.

"Q" is an abbreviation of "Queue", and refers to the time messages wait before being sent. Average, best and worst delays are collected while the run button is held.
The tolerance column refers to the configured link tolerance value. See: Link Tolerances for relevant information.
You may copy the displayed values to the Windows clipboard, or export them to a comma separated values file. The default file name will be "RPCTimes_" followed by a timestamp that identifies when the file was created (not when the values were collected.)

Source Debugger

The Source Debugger is the most powerful and comprehensive of the VTScada debugging tools.



Note: If the Source Debugger application is not visible in the VAM, you must manually add it. The Source Debugger application's directory is

named "Source Debugger," and is stored within the VTScada installation directory.

The Source Debugger enables you to test code written using the VTScada scripting language. Using the Source Debugger, you can:

- Examine the source code for the VTScada application you are debugging.
- Pause execution of VTScada script code by setting breakpoints.
- View the execution history of a specific thread when a breakpoint is "hot".
- View and modify variable values.
- Search for a specific instance of a module by specifying a Boolean expression .
- Locate module instances that use a specific variable in steady-state, and find the module instance that is setting a variable in steady-state.
- Build one to four "Watch" lists of selected variables that remain on view continuously.
- Slay instances of modules.

Note: In addition to the Source Debugger, VTScada includes an older application named "Debugger". While the older debugger can still be useful in certain situations, the Source Debugger is by far the more advanced of the two tools.

Related Information:

...Debugger Utility - used to examine the contents of modules and their properties.

...Source Debugger Components - window by window component list.

...Selecting an Application for Debugging - methods of selecting what to work with.

...Dump Files - creating and viewing.

...Examining Code Paths Using Thread Display - use of the Tread Display window.

...Working with Breakpoints and Data Breakpoints - setting, using, releasing.

...Working with Watches - how to set and release.

...Working with Variables, Arrays, Pointers, Constants, and Parameters - how to examine the various data structures.

...Working with Modules - search, display, refresh, step into, sort and filter.

...Working with the Execution History - a history of threads that have run.

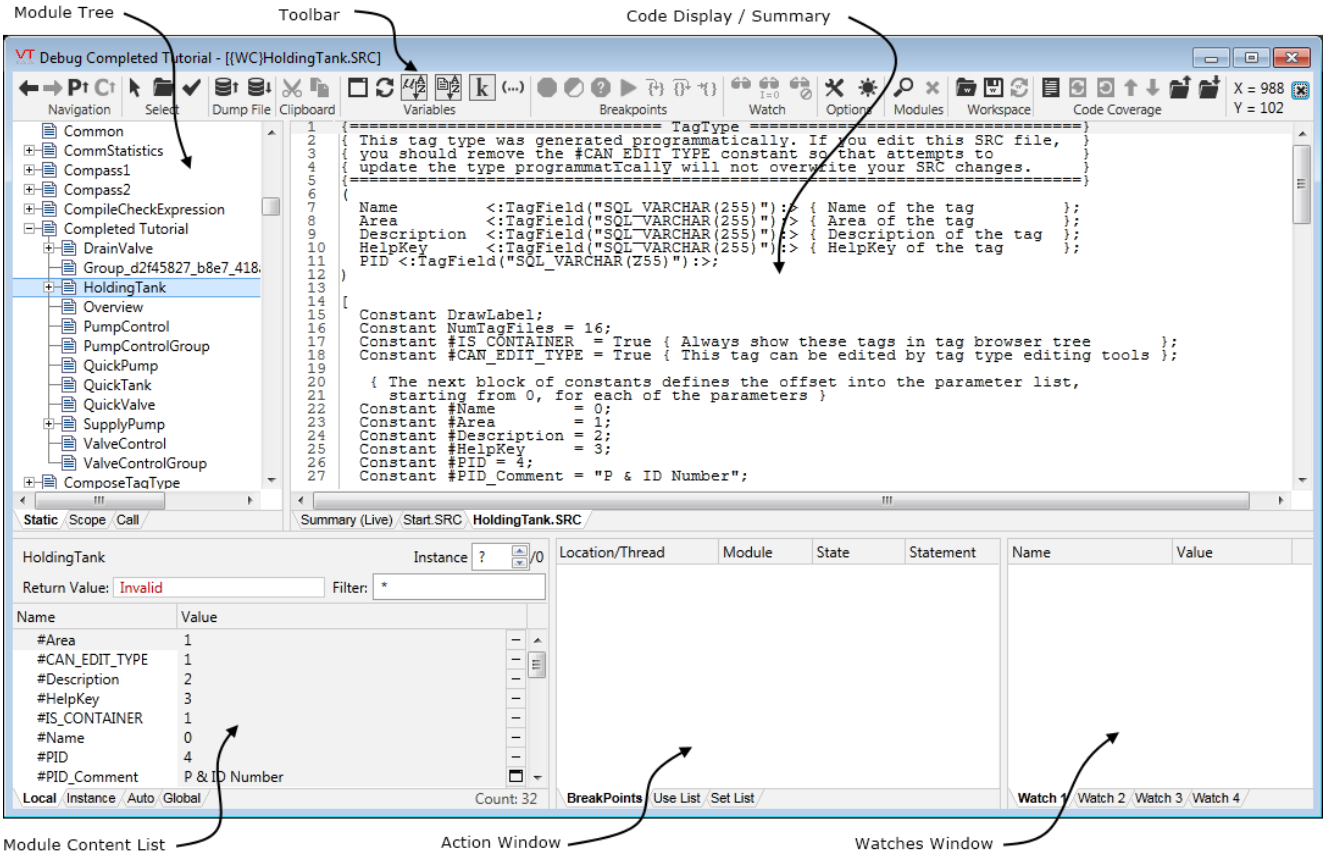
...Copying and Pasting Code Using the Source Debugger - the Source Debugger enables you to copy and paste any kind of object.

...Source Debugger Options - display and control options, including workspace definitions.

...Code Coverage - discover how much of your code actually runs.

Source Debugger Components

The various components of the Source Debugger are as labeled. A description of each can be found in the following topics.



Components:

...Source Debugger: Tool Bar

...Source Debugger: Module Tree

...Source Debugger: Code Display

...Source Debugger: Summary (Live) Tab

...Source Debugger: Summary (Dump) Tab

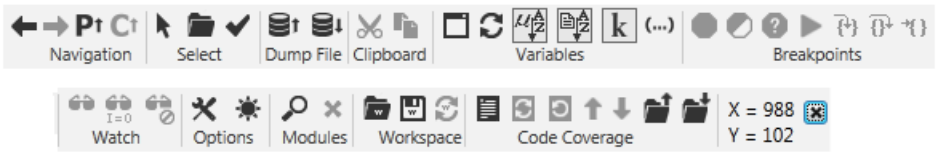
...Source Debugger: Module Content Window

...Source Debugger: Action Window

...Source Debugger: Watch Window

Source Debugger: Tool Bar

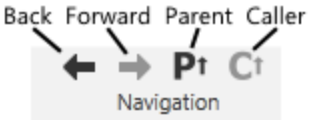
At the top of the Source Debugger appears a tool bar, similar to that shown here (broken into two lines to fit this page).



Note: As with all VTScada buttons and tools, placing your mouse pointer over each button opens a tool tip that indicates the function of the button, along with its keyboard shortcut.

The buttons on the Source Debugger's tool bar are as follows. (Left to right, by group).

Navigation Group



Back Button & Forward Button

Use these two buttons to move back and forth through the modules you have recently viewed. You must have moved backward in the module order for the forward button to be enabled.

Note: You may also use the keyboard combinations Alt + Left Arrow (move left) and Alt + Right Arrow(move right) to step through the modules.

Parent Button

Click the Parent button to move to the parent of the module selected in the module tree.

Caller Button

The Caller button can be clicked to move to the caller of the module selected in the module tree.

Select Group

Select Window to Debug Button

Select a window to debug while your VTScada application is running.
See also: Selecting an Application for Debugging

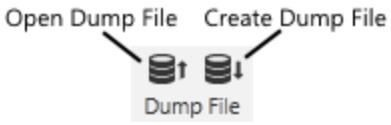
Open Source File Button

The Open Source File button can be clicked to browse to and open a specific source file for debugging.
See also: Open a Source Code File for Debugging

Debug Test Framework Button

Launch the Test Framework application, to run pre-written tests on the selected module.
See also: Test Framework Application

Dump File Group



Open Dump File Button

The Open Dump File button can be clicked to browse to and open a specific dump file for debugging.
See also: Dump Files

Create Dump File Button

The Create Dump File button can be clicked to create a dump file for the application being debugged.
See also: Dump Files

Clipboard Group

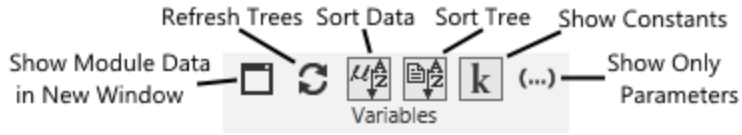
Cut Button

The Cut button can be clicked to cut selected text from the source file being debugged.
See also: Copying and Pasting Code Using the Source Debugger

Copy Button

The Copy button can be clicked to copy selected text from the source file being debugged to the clipboard.
See also: Copying and Pasting Code Using the Source Debugger

Variables Group



Show Module Data in New Window Button

The Show Module Data in New Window button can be clicked to launch a new window that displays the objects associated with a selected module.
See also: Display the Contents of a Module in a Separate Window

Refresh Trees Button

The Refresh Trees button enables you to refresh the module trees, ensuring that any new module instances that have been added are displayed.
See also: Refresh the Module Tree

Sort Data Button

Sort the variables, parameters, and modules in the module content window in ascending order alphabetically.

See also: Sort Data in the Module Tree or Module Content Window

Sort Tree Button

Sort the modules in the module tree in ascending order alphabetically.

Hide Constants Button

Hide any constants that are being displayed in the module content window.

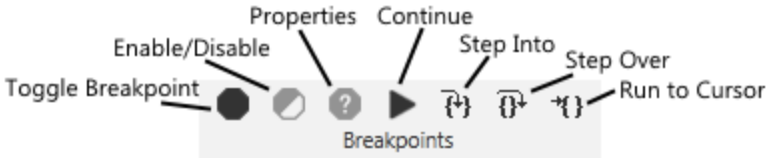
See also: Working with Variables, Arrays, Pointers, Constants, and Parameters

Show Only Parameters Button

Command the Source Debugger to display only parameters for the selected module in the module content window.

See also: Working with Variables, Arrays, Pointers, Constants, and Parameters

Breakpoints Group



Toggle Breakpoint (F9) Button

Set a breakpoint in the code you are viewing. A breakpoint indicates to the Source Debugger that the selected application should pause execution at a certain point.

See also: Working with Breakpoints and Data Breakpoints

Enable/Disable Breakpoint (Ctrl + F9) Button

Disable an existing breakpoint or enable a disabled breakpoint in the code you are debugging or examining.

Breakpoint Properties (Alt + F9) Button

Open a window within which conditional breakpoints can be configured.

Continue (F5) Button

Continue execution of the code you are debugging or examining when a breakpoint has been hit and code execution paused.

Step Into (F11) Button

The Step Into button steps you into a statement, even if it is in another source file.
See also: Step Into a Statement

Step Over (F10) Button

The Step Over button steps you onto the next instruction. Stepping over a subroutine call runs it at full speed and halts execution of the stepped thread at the next function or statement.

Run to Cursor (F12) Button

The Run to Cursor button places a temporary breakpoint at the high-lighted line of source code. If the Source Debugger has halted execution of a thread and that break is highlighted in the Breakpoints tab of the Action window, that thread's execution will be resumed at full speed. The Run to Cursor button can be used like a shortcut that will run from a breakpoint to wherever the highlighted line of code is situated.
See also: Run Code from a Breakpoint to a Selected Line

Watch Group

Add Watch (Ctrl + W) Button

Add the selected variable and its value from the module content window to the selected tab in the watch window.
See also: Setting a Watch on a Variable

Add Watch Expression Button

Add the selected expression from the module content window to the selected tab in the watch window.

See also: Set a Data Breakpoint

Remove Watch (Del)

Remove the selected variable and its value from the watch window.

See also: Remove a Watch

Options

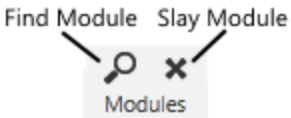
Set several options that affect the way the Source Debugger appears and behaves.

See also: Source Debugger Options

Daytime/Nighttime

Toggles the background between white (daytime) and black (nighttime) mode. The foreground (text) color will be the reverse.

Modules Group



Find Module

Enter a Boolean expression and search to find all instances of the current module.

See also: Search for a Specific Module Instance

Slay

Terminate the execution of the selected module instance.

See also: Slay a Module Instance

Workspace Group

Open Workspace

A workspace is the set of the selected module, the breakpoints, and the watches for a Source Debugger session, stored in a .DWS file. Use this button to re-open a workspace that you have saved with the Save Workspace button.

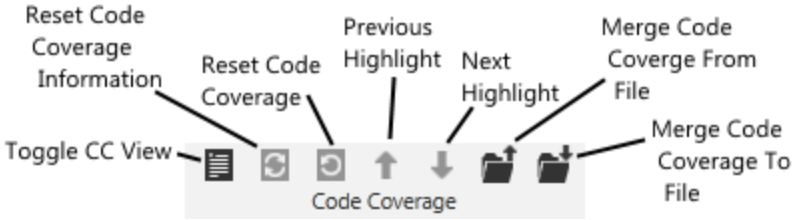
Save Workspace

See preceding description for Open Workspace. Use this button to save the breakpoints and watches, and the selected module that you are working with, from one session to another.

Reload Auto Workspace

If you have enabled the automatic workspace reload feature (done in the Options dialog - see: Source Debugger Options) then you may use this button to load the last automatically-saved workspace.

Code Coverage Group



Toggle Code Coverage

Toggles the display of code coverage (the indicator of what modules, states, statements and parameters used)

Reset Code Coverage Information

Resets the code coverage counters to zero.

Refresh Code Coverage

Displays an up-to-date view of the code coverage.

Previous Highlight

Steps through the display to the previous statement (parameter…) highlighted as having been used by the module.

Next Highlight

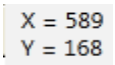
Steps through the display to the next statement (parameter…) highlighted as having been used by the module.

Merge Code Coverage From File

Loads the code coverage record as stored in a file, into the current display. This is especially useful for team debugging where various team members are testing different parts of the code and wish to merge their results.

Merge Code Coverage To File

Saves the code coverage, from the current session to a file. This is especially useful for team debugging where various team members are testing different parts of the code and wish to merge their results.



Mouse Coordinates

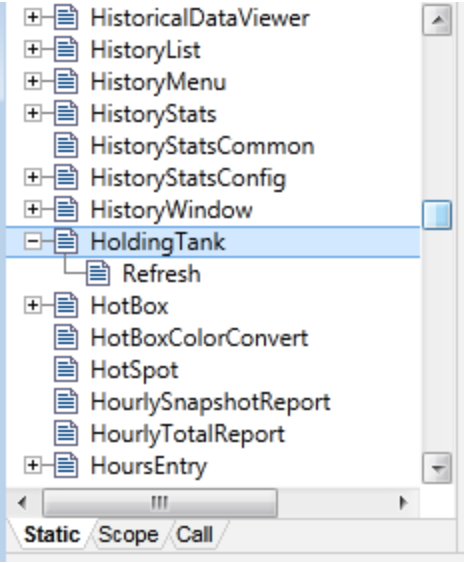
To the far right of the Source Debugger's tool bar is the mouse coordinates section. As you move your mouse pointer around on the screen, it reports the X and Y coordinates with respect to the origin of the VTScada window that the mouse pointer is over.

Close Window

This small "x" at the right-most edge of the toolbar will close the selected window in the Code Display area.

Source Debugger: Module Tree

To the left of the Source Debugger window appears the module tree.



As its name implies, the module tree reports a list of all modules in the selected application, organized in a hierarchical tree structure. The modules shown in the trees are not dynamically updated in real-time. To do so would impose a significant execution overhead that would likely impair or affect the normal running of the system. The trees are automatically refreshed when a breakpoint is "hit" (see Set or Clear a Breakpoint), or when the "Refresh Trees" tool bar button is clicked.

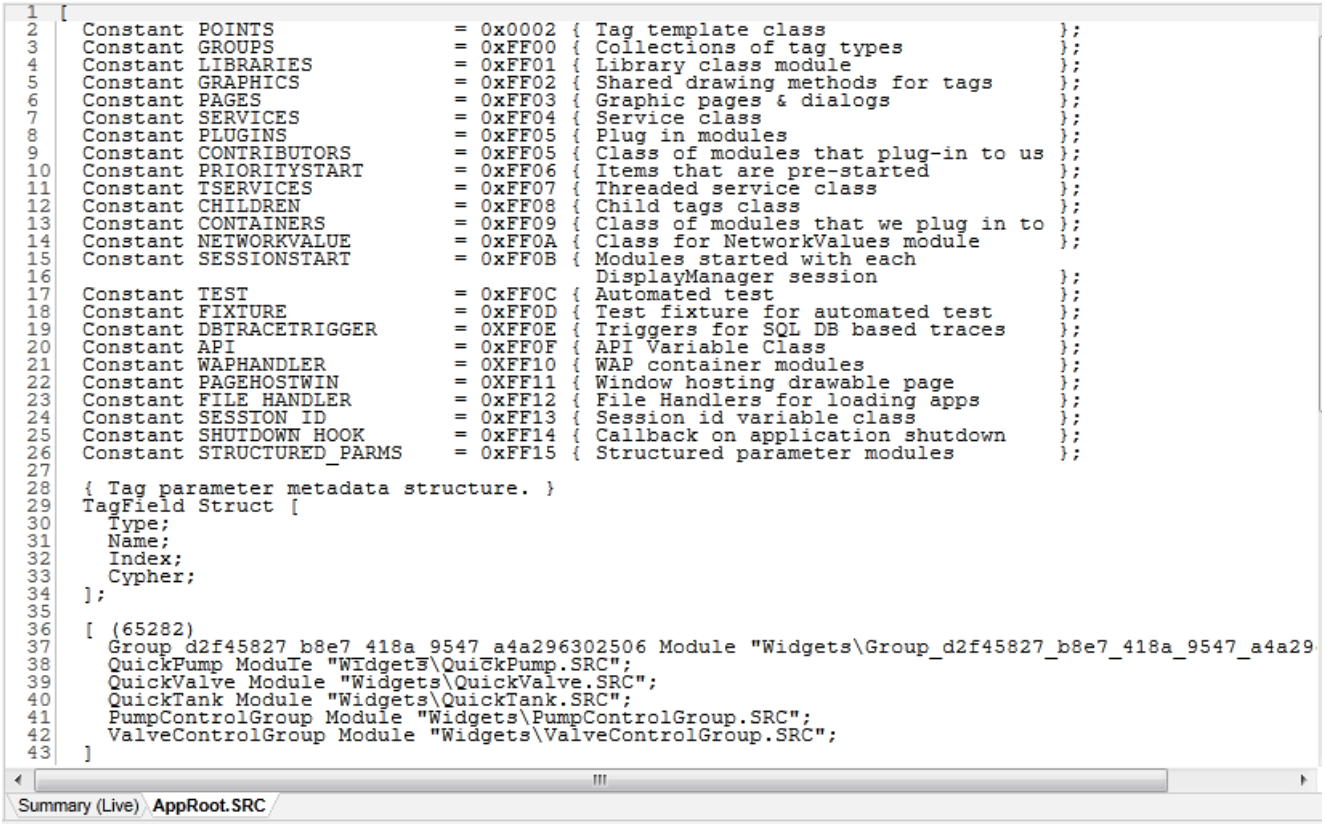
A small square box to the left of the module name indicates a module that has submodules. A plus symbol in the box indicates that display of the submodules is hidden, while a minus symbol in the box is displayed when the submodule tree has been expanded. You may expand any directory to view its sub-directories by clicking the plus sign to its left, or by double-clicking the module name. Likewise, you can contract a branch in the module tree by clicking the minus sign or double-clicking the module name again.

At the bottom of the module tree appear three tabs that may be clicked to change the type of modules that are displayed in the module tree as follows:

Static: Displays static modules. This represents the hierarchy of modules present in the source code of your script application when it was compiled. The static tree shows you exactly what is in the engine, statically loaded without any parent.
Scope: Displays the "parent/child" relationships of each module instance. Whereas the static tree displays one entry per compiled module, the scope tree displays one entry per module instance that is running, therefore, there may be none, or many entries for the same module code. A module instance in the scope tree will resolve its variable references in its parent module instances shown in this tree.
Call: Displays the actual call sequence for each module instance (i.e. which module has called a module instance (see Module Calls)).
See also: Sorting Data in the Module Content Window or Module Content Window.

Source Debugger: Code Display

The code display area occupies the majority of the space in the Source Debugger.



```
[
  Constant POINTS               = 0x0002 { Tag template class                    };
  Constant GROUPS               = 0xFF00 { Collections of tag types              };
  Constant LIBRARIES            = 0xFF01 { Library class module                  };
  Constant GRAPHICS             = 0xFF02 { Shared drawing methods for tags       };
  Constant PAGES                = 0xFF03 { Graphic pages & dialogs               };
  Constant SERVICES             = 0xFF04 { Service class                         };
  Constant PLUGINS              = 0xFF05 { Plug in modules                       };
  Constant CONTRIBUTORS         = 0xFF05 { Class of modules that plug-in to us };
  Constant PRIORITYSTART        = 0xFF06 { Items that are pre-started            };
  Constant TSERVICES            = 0xFF07 { Threaded service class                };
  Constant CHILDREN             = 0xFF08 { Child tags class                      };
  Constant CONTAINERS           = 0xFF09 { Class of modules that we plug in to };
  Constant NETWORKVALUE         = 0xFF0A { Class for NetworkValues module        };
  Constant SESSIONSTART         = 0xFF0B { Modules started with each
                                           DisplayManager session               };
  Constant TEST                 = 0xFF0C { Automated test                        };
  Constant FIXTURE              = 0xFF0D { Test fixture for automated test       };
  Constant DBTRACETRIGGER       = 0XFF0E { Triggers for SQL DB based traces      };
  Constant API                  = 0xFF0F { API Variable Class                    };
  Constant WAPHANDLER           = 0XFF10 { WAP container modules                 };
  Constant PAGEHOSTWIN          = 0XFF11 { Window hosting drawable page          };
  Constant FILE_HANDLER         = 0xFF12 { File Handlers for loading apps        };
  Constant SESSION_ID           = 0xFF13 { Session id variable class             };
  Constant SHUTDOWN_HOOK        = 0xFF14 { Callback on application shutdown      };
  Constant STRUCTURED_PARMS     = 0xFF15 { Structured parameter modules          };

  { Tag parameter metadata structure. }
  TagField Struct [
    Type;
    Name;
    Index;
    Cypher;
  ];

  [ (65282)
    Group_d2f45827_b8e7_418a_9547_a4a296302506 Module "Widgets\Group_d2f45827_b8e7_418a_9547_a4a29
    QuickPump Module "Widgets\QuickPump.SRC";
    QuickValve Module "Widgets\QuickValve.SRC";
    QuickTank Module "Widgets\QuickTank.SRC";
    PumpControlGroup Module "Widgets\PumpControlGroup.SRC";
    ValveControlGroup Module "Widgets\ValveControlGroup.SRC";
  ]
```

Summary (Live) AppRoot.SRC

As its name implies, the code display area shows the source code for the selected modules. Selecting a module from the Module Tree displays the source code file containing that module, positioned at the line in the source file that contains the module code.

At the bottom of the code window appear a series of tabs that mark the source code files that you have viewed using the Source Debugger. A new tab is added each time you click on a module for viewing that is not in a source code file for which a tab exists. You may click any of these tabs to view the associated source code.

The Summary (Live) tab displays details about the live system (see Source Debugger: Summary (Live) Tab). If you are reviewing a crash dump, the label on the tab will change to match (see Source Debugger: Summary (Dump) Tab).

On the left side of the source code, the line number within the source file is displayed. Clicking on a line within the display highlights that line and selects it for the purposes of setting, enabling, or clearing a breakpoint.

Source Debugger: Summary (Live) Tab

The Summary (Live) tab of the code display area can be clicked to display details about the live system.

Note: You may also debug a dump file using the Source Debugger. Please refer to Source Debugger: Summary (Dump) Tab for details.

There are five sections on the System Summary window (each of which can be unrolled or rolled to view more or less information):

- Operating System: Identifies the operating system for the local PC.
- VTScada Information: Provides a snapshot of the VTScada installation running on the local PC, including the amount of memory that is being used. This information is dynamically updated. Unrolling the VTScada Information section enables you to view basically the same information that is contained in the About dialog).
- Version number
- Serial number
- License type
- Free updates expiry (extracted from the installation key)
- Max tags allowed
- Max browser clients allowed
- Alarm Notification System installed?
- Memory used
- Applications: Provides information about the applications installed on the local machine, including a list of those running and those that are stopped. You may expand the Running and Stopped subsections to see the names of the applications. The count (to the right of the Applications heading) includes applications that haven't been added to the VAM. Please note that the System Library is always running so that you may debug the VAM.
- Modules: Provides details similar to Instance Count, except more efficient. When the Modules tab is unrolled, Instance Count is run. This is a fairly expensive process in terms of system resources, so this section only updates periodically unless it is unrolled again.

- Threads: Identifies the set of running threads within VTScada. The number to the right of the thread name is the Windows thread identifier. The module that last ran, its state, and its statement number are identified. This section is updated on a fairly slow cycle to save system resources, but you may use the update button for faster updates. Under the System subheading (within the Threads section) the sequence of execution is provided (this is the same information as is presented in the Thread List Application – time, module name, state, and statement number).
- Execution History: Displays a history of all threads that have run, sorted by time. Of use especially when debugging a source dump as you can quickly find what modules or statements ran just before the dump occurred. See: Working with the Execution History.

Source Debugger: Summary (Dump) Tab

The code display window can display the contents of a dump file. At first glance, this may appear to be the same as for the Live tab however, there are several small but important differences.
When a dump file is displayed, the tab will be labeled, "Summary (Dump)". All information in the Summary (Dump) window applies to the PC on which the dump occurred, rather than to the local PC (unless the dump occurred on the local PC). (Information on debugging a live system is provided in Source Debugger: Summary (Live) Tab.)

> Note: You may generate a dump file at any time. Instructions appear in Generating a Dump File.

At the top of the Summary (Dump) window appears the text "System Summary", beneath which the path to the dump file being examined is displayed.
There are five sections on the System Summary window (each of which can be unrolled or rolled to view more or less information):

- Operating System: Identifies the operating system for the PC on which the dump took place.
- VTScada Information: Provides a snapshot of the VTScada installation running on the PC on which the dump took place, including the amount of

memory that was being used. (This information is static, as a dump file is being examined, rather than a live system.) Unrolling the VTScada Information section enables you to view basically the same information that is contained in the About dialog).

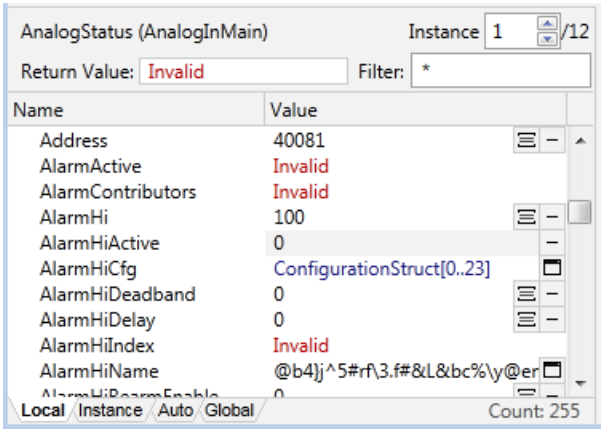
- Version number
- Serial number
- License type
- Free updates expiry (extracted from the installation key)
- Max tags allowed
- Max browser clients allowed
- Alarm Notification System installed?
- Memory used
- Applications: Provides information about the applications installed on the PC on which the dump took place, including a list of those that were running and those that were stopped at the time of the dump. You may expand the Running and Stopped subsections to see the names of the applications. The count (to the right of the Applications heading) includes applications that weren't added to the VAM at the time of the dump.
- Modules: Unlike the Modules section for a live system summary (see Source Debugger: Summary (Live) Tab), no modules will be displayed in the Modules section for a dump file, as none will be loaded and running.
- Threads: Identifies the set of threads that were running within VTScada at the time of the dump. The number to the right of the thread name is the Windows thread identifier. The module that last ran, its state, and its statement number are identified. Under the System subheading (within the Threads section) the sequence of execution that last took place prior to the dump is provided (this is the same information as is presented in the Thread List Application – time, module name, state, and statement number).

Note: When examining a dump file, the last statement executed can be determined from the thread list.

Note: You may point the Source Debugger to source code on another PC if you are debugging on a PC without source code. To do so, please read Source Debugger Options.

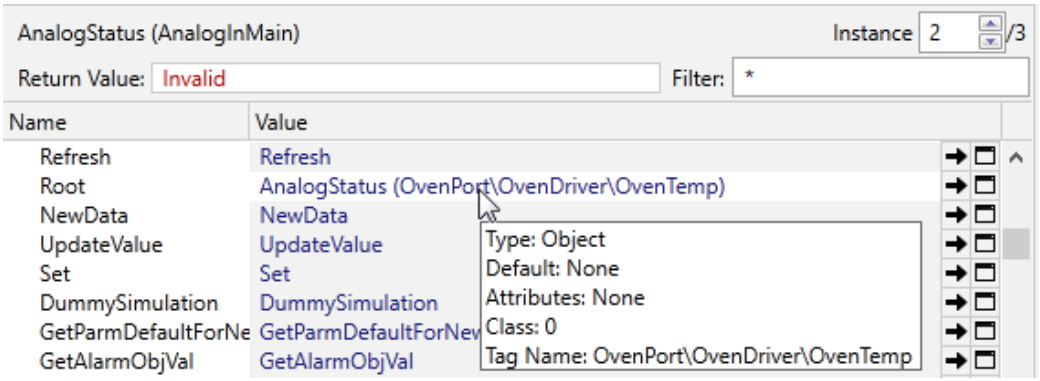
Source Debugger: Module Content Window

The lower left portion of the screen is occupied by the module content window.



This window displays a list of the contents of the module selected in the module tree, including its variables, parameters, and related sub-modules. The count at the lower right corner shows the number of variables within the filtered list.

If the module you are examining is a tag, you can discover the name of each instance by searching for the Name property, or by looking for the Root property. The tag name will be shown in brackets following the value of Root, and will be shown if you hover over the property.



Above the module content window appears the name of the module being viewed, along with the total number of instances of that module, and the instance you are viewing. The up and down arrow buttons next to the current instance number can be used to scroll through the instances of the selected module. Alternatively, you may click in the

current instance number box and type in an instance number. When a new module instance is selected, the scope and call module trees will change to show the hierarchical position of the selected module instance.
The Return Value field displays (and enables you to set) the return value for the module being viewed, while the Filter field enables you to filter the contents of the window to view only those modules that match the filter criteria (see Filtering Data in the Module Content Window). The Return Value's edit field will be disabled if it can't be set.
The leftmost column of the variable/value list displays the name of each variable, parameter, or submodule within the displayed module. The rightmost column displays the current value of that variable for the selected module instance. You can change the value of any variable (see Changing the Value of a Variable).
A variety of tools are available to help you further examine each variable and module in the window:

There are three tabs featured at the bottom of the module content window:
Local: The local tab displays only the variables and values that are local to the selected object or module.
Instance:
Auto: The Auto tab displays the variables and values near the selected source code line. (Near is defined as one source code line before and one after the selected one.) This works both when simply clicking on a line of source code and is perhaps of most use when single stepping through lines of code. The Auto tab will automatically refresh as you step through the code. Automatic scope resolution will walk up through the code to find array declaration and displays in the Auto tab window.
Global: The Global tab displays the contents of the local tab, plus all variables and values in the scope of the selected object or module (see Displaying Objects with a Global Scope in the Module Content Window).

Related Information:

...Switch to module

...View contents

...Convert number

...View metadata

Switch to module

Module and object values have a right-facing arrow button next to them.  Clicking on the arrow button or double-clicking on the value of the variable will cause the module or module instance addressed by the value to become the selected one.

View contents

Array and pointer values have a button with a small window icon next to them.  Clicking on that button or double-clicking on the value of the variable will cause a pop-up window to be displayed that enables you to view the content of arrays and follow pointers (see Displaying Array and Pointer Contents). In the case of modules, clicking that button will open a secondary window in which the details about the module's contents will be displayed (see Displaying the Contents of a Module in a Separate Window).

If you were reading the last two paragraphs carefully, you might now be wondering whether double-clicking on the value of the variable selects a module instance or if it opens a pop-up. The answer is that it does both.

Convert number

Any numeric quantity can be converted to octal, decimal, hex, floating point, or timestamp from its radix. To cycle through these numeric conversions, simply continue to click the button labeled with a dash to the right of the variable or parameter  (see Cycling Through Numeric Conversions for a Variable).

View metadata

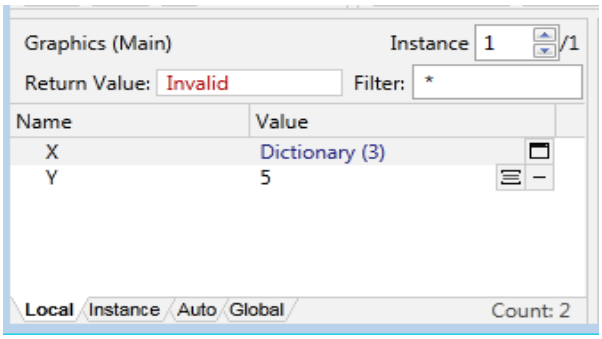
Any value that has underlying metadata will have a metadata button beside it.  (See Dictionaries) Clicking this button will open a pop-up window that enables you to view the variable's metadata.
The metadata button should not be taken as indicating whether or not a variable is a dictionary. For example, a dictionary created with several name/value pairs, but without a root value, will not have a metadata button.
A value in VTScada has properties, other than just its value. By resting the mouse pointer over a value in the module content window, a tool tip window is displayed for several seconds, listing the properties of the selected value. For example:

Type: Short
Default: None
Attributes: Retained
Class: 0

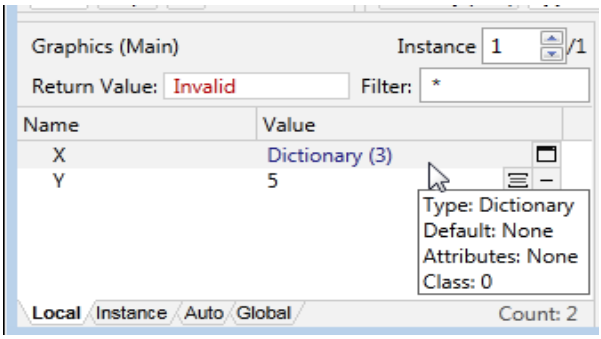
The rules for how dictionaries are displayed in the module window can be confusing to new users. The following example may help to at least illustrate the rules:
Given the following code that creates two dictionaries; the first with no root value and the second with a root value of 5:

```
If Watch(1);
[
  X = Dictionary();
  X["A"] = "Data A";
  X["B"] = "Data B";
  X["C"] = "Data C";

  Y = Dictionary(1, 5);
  Y["A"] = "Data A";
  Y["B"] = "Data B";
  Y["C"] = "Data C";
]
```

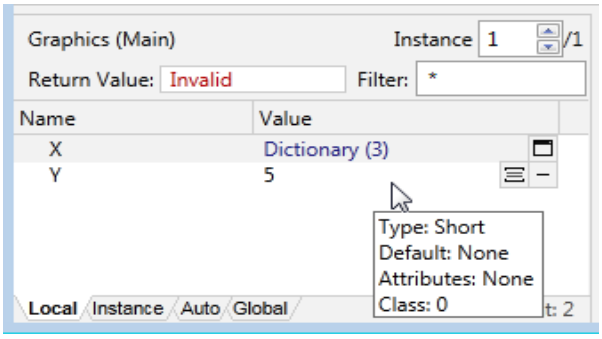
The module content window will look like the following:



Hovering over each of the variables in turn will produce tool tips as shown next:



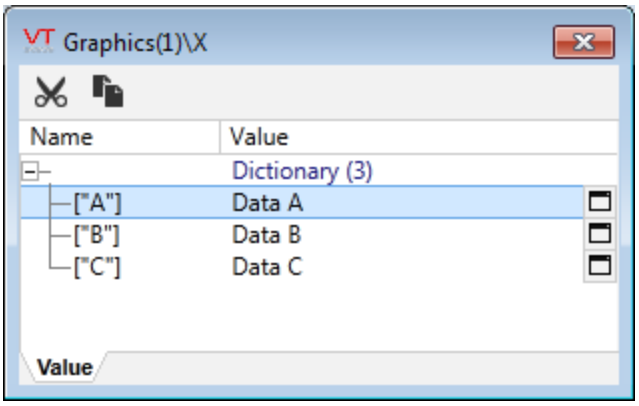
Tooltip associated with a dictionary having no root value



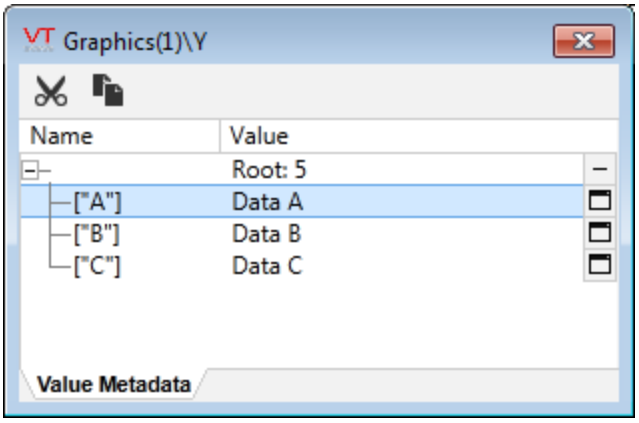
Tooltip associated with a dictionary having a root value

Note that only Y, which has a root value as well as metadata gets a metadata button.

X gets the Window button, which in this case performs a somewhat similar function…

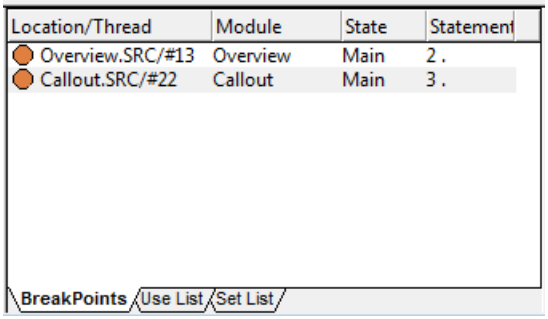


Value of Y from the metadata button



Source Debugger: Action Window

The action window is the middle window appearing at the bottom of the Source Debugger.



The functionality of the action window is dependent upon the tab that is selected beneath it. These tabs and their purpose are:

BreakPoints: The BreakPoints tab displays a list of the set breakpoints, including the source file location, module, state, and statement in which

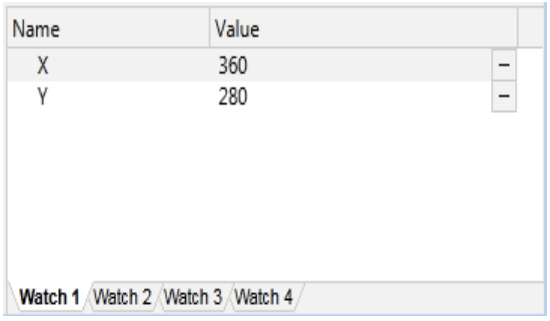
they exist. The breakpoints tab also displays any "hit" breakpoints and the running thread that has been paused.

Use List: The Use List tab displays the source file location of each running steady-state statement that depends on the value of the variable selected in the module content window (i.e. the "consumers" of the value). Each statement so referenced will be triggered for execution when the contents of the variable instance changes.

Set List: The Set List tab displays the source file location of each running steady-state statement that is setting the value of the variable selected in the module content window (i.e. the "producer" of the value). In correctly written code, there should only be one such reference. Multiple such references result in a "double-set" condition and yields an invalid value for the variable instance.

Source Debugger: Watch Window

The Source Debugger enables you to select many variables from different module instances and add them to one of the four Watch tabs within the watch window. This enables you to collect together variables from many different module instances and be able to view them collectively, regardless of which module is selected. The variable/value pairs placed in the watch window have exactly the same format and capabilities as variable/value pairs in the module content window.



Selecting an Application for Debugging

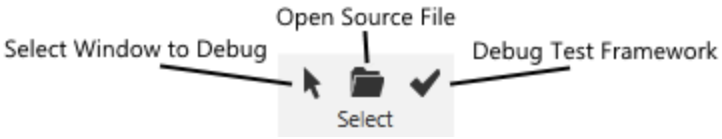
There are two ways to select applications for debugging:

Click the application's name/link in the Applications section of the System Summary (see Source Debugger: Summary (Live) Tab).

OR

Within the VTScada application, navigate to the page whose source code you wish to examine.

In the Source Debugger, click the Select Window to Debug.



Click on the appropriate window in your VTScada application. The Source Debugger will select the running module instance that contains the graphic statement under the mouse pointer as the current module instance, and will display the module trees, content list, and source code for that module window.

You can select an application that is not running using the first option (above). Simply click the name of the application in the Stopped section of the Applications section of the System Summary. You will not see modules listed in the Modules section, as none will be loaded and running. You can place a break point on a non-loaded application, and then run it, however, you must generate debug symbols for this to occur.

Each time you navigate to a different module in the module tree, or select a new window in your VTScada application for debugging, the Source Debugger will change the selected module instance in the module trees, module content window, and code display. If the source file containing the selected module has not yet been displayed by the Source Debugger, a new tab will be added to the source code window and made the current tab. You may click any tab at any time to view the code for a different source code file. You may use the forward and back tool bar navigation buttons, or the corresponding keyboard shortcuts to move backwards and forwards through the module instances that you have selected and viewed.

Each application runs in its own debugger section. The title bar identifies which is being debugged. The Source Debugger runs one session per application or crash dump file being debugged. Once the application has

been selected, the session is committed. There is a new, independent session for each.

Related Tasks:

...Open a Source Code File for Debugging

Related Information:

...Editing Code and Recompiling

Open a Source Code File for Debugging

The Source Debugger can be used to open a source code file for debugging, as follows:

1. Click the Open Source File button. The Open Source File dialog will open.
2. Navigate to the application directory containing the source code file you wish to debug.
3. Select the source code file and click the Open button (or double click the selected source code file). The file's source code will be displayed in the code display window.

Editing Code and Recompiling

When you edit code, stop the application, and compile, the Source Debugger updates the source code display and the breakpoint will stay on the statement on which it was set, rather than the line (in case a statement or module was deleted). If you delete the module within which a breakpoint is set, it will disappear.

Dump Files

You can generate a dump file as a snapshot of the current system. To generate a dump file:
Click the Create Dump File button

1. Browse to the location where you wish to save the dump file.
2. Enter a name for the dump file in the File Name field.
3. Click the Save button. The dump file will be generated and saved in the specified location.

Viewing the Contents of a Dump File

Note: Before you can view the contents of a dump file, you must first configure the Symbol Server tab of the Source Debugger's Options dialog.

The Source Debugger can do all the same things with a dump file that you can do with a live system. To open a dump file for debugging:

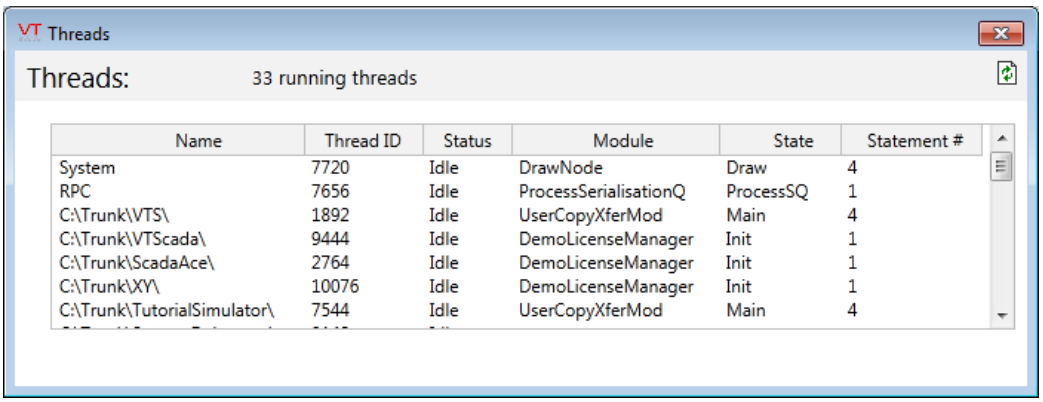
1. Click the Open Dump File button. The Open Dump File dialog will open.
2. Browse to and select the dump file you wish to analyze.
3. Click the Open button.
   The selected dump file will open in the code display window in its own separate tab labeled "Summary (Dump)" (see Source Debugger: Summary (Dump) Tab).

Examining Code Paths Using Thread Display

You can use the thread display to examine code paths. To launch the thread display:

1. Click the Summary (Live) tab of the Code Display window.
2. Click the New Window button to the right of the Threads subsection.

The thread display will launch in a separate window, as shown:



At the top of the thread display appears a count of the total number of running threads for the application being debugged. A summary of data for each thread then appears as a list in the thread display. This data includes:

- The name of each thread.
- The name of the module to which the thread belongs.

- The state for each thread.
- The number of statements associated with each thread.

Working with Breakpoints and Data Breakpoints

Programmers use breakpoints to test and debug programs by causing the program to stop at scheduled intervals so that the status of the program can be examined in stages. Breakpoints (and watches) can be saved from one debugging session to another using the Save Workspace and Open Workspace buttons.
There are two types of breakpoints that are referred to throughout this document:
Breakpoint: Wherever the term "breakpoint" is used, code breakpoint should be inferred. A code breakpoint is a programmer-defined break in code that when reached triggers a temporary halt in the program. To set a breakpoint, please refer to Setting a Breakpoint.
Data Breakpoint: A data breakpoint is a breakpoint set on data that triggers a break when the data is set or changed. To set a data breakpoint, please read Setting a Data Breakpoint.

Clearing a Breakpoint

Clicking the Toggle Breakpoint button will clear a selected breakpoint and remove the marker.

Related Information:

...Setting a Breakpoint

...Conditional Breakpoints

...Examining State at a Breakpoint

Related Tasks:

...Set a Data Breakpoint

...Run Code from a Breakpoint to a Selected Line

...Enable or Disable a Breakpoint

Setting a Breakpoint

A breakpoint is a set location in source code that indicates to the Source Debugger that any thread executing that source code is to be paused immediately prior to executing it (see also Data Breakpoints).

> Note: If your application uses multiple threads, it is quite possible for different threads to execute a given statement. This might typically be the case where the breakpoint was placed in a subroutine that could be called by modules running on different threads.

Placing a breakpoint freezes the state of the application being debugged at a specific point so that you may examine it.

To set a breakpoint:

Select the line of code at which you wish to place the breakpoint (line selection is keyboard and mouse driven). A breakpoint indicator will appear at the selected line and the breakpoint window will reflect the information.

What Happens Once a Breakpoint Has Been Set?

Application execution will continue normally at full speed until a statement that contains a breakpoint is about to be executed. At that point, execution of the thread that is attempting to execute the statement containing the breakpoint will be suspended and another entry will be made in the breakpoint tab of the Source Debugger: action window, listing the thread that has been suspended and the breakpoint that has been hit. The breakpoint tab will be forced to the front of the action window, the tab containing the source code with the breakpoint will be forced to the front of the code display window, and the Source Debugger window itself will be made the active window. Additionally, the module trees will automatically be refreshed and positioned at the module instance that contains the breakpoint. This means that you can see a "snapshot" state of the module trees at the time that the breakpoint was encountered.

The breakpoint indicator will change to show that it has been "hit" by displaying a yellow arrow over its top:


What Happens to Other Threads Once a Breakpoint Has Been Set?

Note that only the thread that executes a statement containing a breakpoint will be suspended. Other threads within VTScada, including those within your application, will continue to run normally. If this were not the case, you wouldn't be able to use the Source Debugger, as it is merely a script application. The implication of this is that while one thread is paused on a breakpoint, another thread may attempt to execute the same statement, and hence cause another breakpoint. This would result in exactly the same sequence of events, with a second entry showing a hit breakpoint being made in the breakpoints tab of the action window. While a thread is paused at a breakpoint, you may navigate to any module in any of the module trees that you wish, and examine or modify the contents of any data variable. In other words, suspension of execution of a thread affords you the time to have a "look-around" at the state of things at a given point in the execution of your application. This can be most useful in detecting transitional states that would otherwise be difficult to find.

How Do I Continue Execution Past a Breakpoint?

To continue execution past a breakpoint, ensure that the breakpoint from which you wish to continue execution is selected in the breakpoints tab of the action window, and then click the Continue button. The selected "hit" breakpoint entry in the breakpoints tab of the Action Window will be removed (the breakpoint will remain, but the breakpoint event will be removed). If there are multiple breakpoints hit, only the breakpoint event selected in the breakpoints tab will be continued.

What Happens If a Critical Section is Hit?

A breakpoint can be placed on any source code line whose module has been compiled with debugger symbols (see the "Note" in Run the Source Debugger). This includes scripts that contain critical sections (see CriticalSection). When a breakpoint within a CriticalSection is hit, the CriticalSection is effectively released and re-acquired when you continue execution from that point. This enables other VTScada threads to run, including the Source Debugger. Depending on the code contained within the CriticalSection, this may cause behavioral differences within your application.

Related Tasks:

...Set a Data Breakpoint

Set a Data Breakpoint

To set a data breakpoint:

1. Select the data variable on which you wish to set the data breakpoint.
2. Click the Add Watch Expression button on a Variable.

The Breakpoints tab in the action window will display the thread on which the data breakpoint occurs.
A conditional breakpoint can be set on an array, but won't work with a pointer to a value or a dynamically allocated array.

Set a Breakpoint to Halt Execution When Data is Set

You can set a data breakpoint to halt execution when data is *set*. As soon as an attempt is made to set the value of a data variable, execution will halt. To continue from a data breakpoint, click the Continue button or press F5.
The next attempt to set the variable will be highlighted in the code.

Setting a Breakpoint to Halt Execution When Data is Changed

As soon as an attempt is made to *change* the value of a data variable, execution will halt, unless the data variable is being set to the same value.

To continue from a data breakpoint, click the Continue button or press F5. The next attempt to change the value of the data variable will be highlighted in the code.

Related Information:

...Setting a Breakpoint

Related Tasks:

...Set a Watch

Conditional Breakpoints

You can apply an expression to apply to any breakpoint, either code breakpoints or data breakpoints.

1. Click the Breakpoint Properties button.
   The Breakpoint Properties dialog will open.
   
2. Select the Conditional Breakpoint check box.
3. Click OK.
   A question mark will appear in the breakpoint's symbol in the Source Debugger: action window and in the code display, and a field will open to allow you to enter an expression in the Action window.
4. Enter a valid expression in the field (such as \Name == "AI20_1").
5. Click OK to the right of the field in the Action window (Breakpoints tab).

Note: You must use the scope resolution operator (\) at the beginning of the expression, or an "unknown variable or function" error will result.

Examining State at a Breakpoint

Once you've set a breakpoint, the important thing to do is to look at the data. The Global tab of the module content window displays the name of the variable, data type, default values, and attributes class.

Note: In the case of multidimensional, dynamic, and static arrays, you can view array values in a separate window (see Displaying Array and

Pointer Contents) You may also open a separate window for a selected module. There is no limit as to how many data windows you can have open to enable you to quickly navigate source files.

Related Information:

...Working with Variables, Arrays, Pointers, Constants, and Parameters - displaying Array and Pointer Contents.

Enable or Disable a Breakpoint

To disable a set breakpoint:

1. Click on the breakpoint or data breakpoint within the source code or the breakpoints tab of the Source Debugger: Action window.
2. Click the Enable/Disable Breakpoint button.
   When a breakpoint is disabled, the entry in the breakpoints tab of the Action Window is not removed, but in conjunction with the indicator in the code window, changes in appearance:

This will prevent a breakpoint from suspending execution when the statement containing it is executed, but it makes it easy to locate the breakpoint within the source code. Double-clicking the disabled entry within the breakpoints tab positions the code window at the source code corresponding to the breakpoint.
If the breakpoint being disabled has suspended threads waiting (i.e. it has been hit and not yet continued past), then disabling the breakpoint releases any waiting threads.

To enable a disabled breakpoint:

1. Click on the breakpoint within the source code or the breakpoints tab of the Source Debugger: Action window.
2. Click the Enable/Disable Breakpoint button.

The breakpoint indicators for the selected breakpoint in both the code window and the breakpoints tab will change in appearance to signify an enabled breakpoint.

Run Code from a Breakpoint to a Selected Line

To run code from a breakpoint to a highlighted line of code, click the Run to Cursor button. This operation can be useful when a breakpoint is at the beginning of a source file and the highlight is at the end.

Working with Watches

The Source Debugger enables you to watch variables. You may select many variables from different module instances and add them to one of the four Watch tabs that appear within the watch window. This enables you to collect together variables from many different module instances and view them collectively, regardless as to which module is selected. Additionally, any expression may be watched.
Watches (and breakpoints) can be saved from one debugging session to another using the Save Workspace and Open Workspace buttons.

Related Tasks:

...Set a Watch

...Remove a Watch

Set a Watch

To watch a variable:

1. Select the variable in the module content window whose value you wish to monitor.
2. Select the watch tab to which you wish to add the selected variable. (There are four watch tabs situated below the watch window.)
3. Click the Add Watch button. The variable will be added to the Watch window so that you can monitor it.

To watch an expression:

The Source Debugger enables you to set a conditional watch using an expression. To do so:

1. Select the watch tab to which you wish to add the watch. (There are four watch tabs situated below the watch window.)

2. Click the Add Watch button. A field will open on the selected watch tab to allow you to enter the expression to be watched.
3. Enter an expression (e.g. \I + 42).
4. Press the Enter key. The expression will be added to the selected watch tab.

Note: You must use the scope resolution operator (\) at the beginning of the expression, or an "unknown variable or function" error will result.

Remove a Watch

To remove a watch set on a variable or expression:

1. Select the variable or expression whose value you no longer wish to monitor in the watch window.
2. Click the Remove Watch button.

The variable or expression will be removed from the watch window.

Working with Variables, Arrays, Pointers, Constants, and Parameters

The Source Debugger enables you to display the contents of arrays and pointers in a separate window. To open a secondary window displaying array or pointer contents, click the button with the window icon that appears to the left of the array or pointer reference in the module content window.

Note: Pointers are labeled "Pointer" in green text in the Value column of the module content window, while arrays are labeled "Array" and the number of array elements is indicated.

The secondary window for an array displays the name of the array (e.g. LookupArray), to the right of which the number of array elements is indicated (e.g. Array [0..64]). Beneath the array name/elements count, each array element is listed, starting at 0, and the value for each element is listed in the Value column to its right.

The secondary window for a pointer displays the name of the pointer (e.g. RPCStatus), to the right of which the text "Pointer" appears. Beneath

"Pointer" in the Value column is the value being pointed to by the identified pointer.

Displaying Parameters in the Module Content Window

The Source Debugger enables you to set the module content window to display parameters only. To do so, click the Show Only Parameters button.

Displaying Objects with a Global Scope in the Module Content Window

The Source Debugger enables you to set the module content window to display global variables, parameters, and modules by clicking its Global tab.

Global variables, parameters, and modules are those that can be accessed from the current module.

Hiding Constants in the Module Content Window

If you do not wish to view constants in the objects displayed in the module content window, you may hide them by clicking the Hide Constants button.

Cycling Through Numeric Conversions for a Variable

Any numeric quantity can be converted to octal, decimal, hex, floating point, or timestamp from its radix.

To cycle through these numeric conversions for a variable:

1. Select the variable in the module contents window.
2. Click the button labeled with a dash to the right of the variable or parameter. The label of the button will change to reflect to the numeric conversion being applied to the variable.

Changing the Value of a Variable

The Source Debugger enables you to change the value of any variable in the module content window.

To change the value of a variable:

1. Double-click the variable in the module content window. An edit field will open to the right of the selected variable.
2. Enter a new value for the selected variable in the edit field.
3. Press the Enter key to input the value.

Although you may modify the value of a variable, you may not modify the value of a parameter, a variable that is being set in steady state, or a variable that has a representation that cannot be typed at the keyboard (for example, you cannot modify a variable value that contains an object value). Any value that you cannot modify is disabled.

Related Information:

...Source Debugger: Module Content Window

Working with Modules

To view the source code for a selected module:

- Locate in the module tree the module you wish to debug.
- Click the module to view its source code.

Each time you navigate to a different module in the module tree, or select a new window in your VTScada application for debugging, the Source Debugger will change the selected module instance in the module trees, content list, and source code window. If the source file containing the selected module has not yet been displayed by the Source Debugger, a new tab will be added to the source code window and made the current tab. You may click any tab at any time to view the code for a different source code file. You may use the forward and back tool bar navigation buttons or the corresponding keyboard shortcuts to move backwards and forwards through the module instances that you have selected and viewed.

Related Tasks:

...Display the Contents of a Module in a Separate Window

...Search for a Specific Module Instance

...Slay a Module Instance

...Refresh the Module Tree

...Step Into a Statement

...Step Over Code

...Sort Data in the Module Tree or Module Content Window

...Filter Data in the Module Content Window

Display the Contents of a Module in a Separate Window

The Source Debugger enables you to view the contents of a module (i.e. its variables, parameters, arrays, etc.) in a separate window, which is similar in appearance to the module content window.

1. Select the module whose contents you wish to view in a separate window.
2. Click the Show Module Data In New Window button.

A separate window will open and display the contents of the module.

Related Information:

...Source Debugger: Module Content Window

Search for a Specific Module Instance

You can enter a Boolean expression using the VTScada script syntax. This Boolean expression will be evaluated in the scope of each instance of the selected module. Each module for which the expression evaluates to true is made the selected module. Use this feature to locate a specific module instance among many.

To search for a specific instance of a module:

1. Click the Find Module button. The Find Module dialog will open.
2. Enter a Boolean expression in the field provided.

Note: The Boolean expression must conform to the VTScada script syntax, and must precede all variable references with a scope resolution operator (\).

3. Click the Find button at the top of the Find Module dialog.

This facility can be extremely useful where you are diagnosing a problem in a large application that may have many instances of the same module (such as a tag), and you need to locate a specific instance. As an example, let's assume that you are using the "Completed Tutorial

Example" application that is provided with your VTScada installation, and you wish to locate the instance of the module "AnalogInput" that corresponds to the tag "AI20_2". This module instance can be identified easily enough because you know that the variable "Name" within that module holds the tag's name. As a result, you would search every instance of the module "AnalogInput" for variable Name equals AI20_2. The script expression to do this is:

```
\Name == "AI20_2"
```

Note that the text value of the name is contained within quotes, as per a normal string literal, and that the variable "Name" is preceded by a scope resolution operator (\).

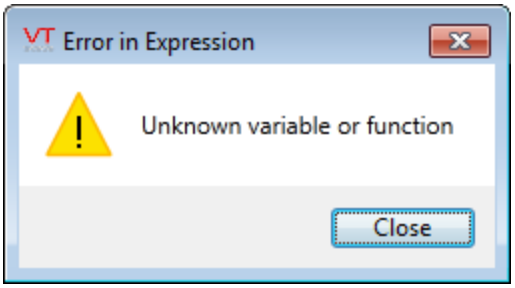
Once you have selected any instance of the module "AnalogInput" and entered the above expression, pressing the Find button compiles the expression on the fly, and then evaluates it within each instance of the selected module. When the expression returns true, the search is stopped and the located module instance is made the selected module instance. Pressing the Find button again causes the search to resume from the point at which it left off.

When no more instances that evaluate to true can be located, a dialog will be displayed.

If the Source Debugger is unable to compile the supplied expression, an error dialog is displayed, describing the error. For example, if the above expression was erroneously typed as

```
(\Name == "AI20_2"
```

the Source Debugger will display a dialog that describes the nature of the error.



Slay a Module Instance

The Source Debugger enables you to slay a module instance.
To slay a module instance:

1. Select the module for which you wish to slay an instance in the module window. The module content window in the Module Content Window displays the number of instances of the selected module that are running in its top right corner.
2. Scroll through the module instance numbers using the Instance spin box until you locate the instance you wish to slay. (Alternatively, by selecting a specific module instance from the scope or call tabs of the module window, you can go directly to a specific instance.)
3. Click the Slay button. A dialog will open and request confirmation that you wish to slay the selected module instance.
4. Click the Yes button. The instance of the module you have selected will be slain.

Related Information:

...Source Debugger: Module Content Window

Refresh the Module Tree

When you have added (or deleted) modules from an application that you are debugging, you can refresh the module tree to ensure you are viewing the application's latest data.
To do so, click the Refresh Trees button.

Step Into a Statement

You can step into a statement, even if it is located in a source file other than the one you are debugging. To do so, click the Step Into button (or press F11).

Note: Two threads will cause multiple breaks; only the one you've selected will be affected.

Step Over Code

Use the Step Over button (or pressing F10) to step on to the next function compute method, except that stepping over a subroutine call runs the call at full speed and halts execution of the stepped thread at the next function or statement.
Stepping causes a break to occur at:

- The next top-level statement executed on the same thread
- The next statement in an IfThen, IfElse, WhileLoop, DoLoop, Case, or Execute clause

Note that b) means that stepping source code written as:

```
1: IfElse(A > B,
2:   Execute(
3:     X = Y;
4:   );
5: { else } Execute(
6:     Z = Y;
7: ));
```

(which is not the normal style of a VTScada programmer) would result in a step from line 1 to line 2 to line 3, or from line 1 to line 5 to line 6.
The same code in the normal style of a VTScada programmer:

```
1: IfElse(A > B, Execute(
2:     X = Y;
3:   );
4: { else } Execute(
5:     Z = Y;
6: ));
```

The preceding example would result in a step from line 1 to line 1 to line 2, or from line 1 to line 4 to line 5. On line 1, the first step stops on the execute, hence the apparent non-movement of the source line when stepping. In such cases, the Breakpoint tab of the Action window will show you the parameter of the statement on which you are stopped.

Sort Data in the Module Tree or Module Content Window

To alphabetically sort the data displayed in the module content window in ascending order (A-Z), click the Sort Data button in the tool bar.
To alphabetically sort the data displayed in the module tree in ascending order (A-Z), click the Sort Tree button in the tool bar.

Filter Data in the Module Content Window

The Source Debugger enables you to filter the data being displayed in the module content window.
To filter the contents of the module content window:

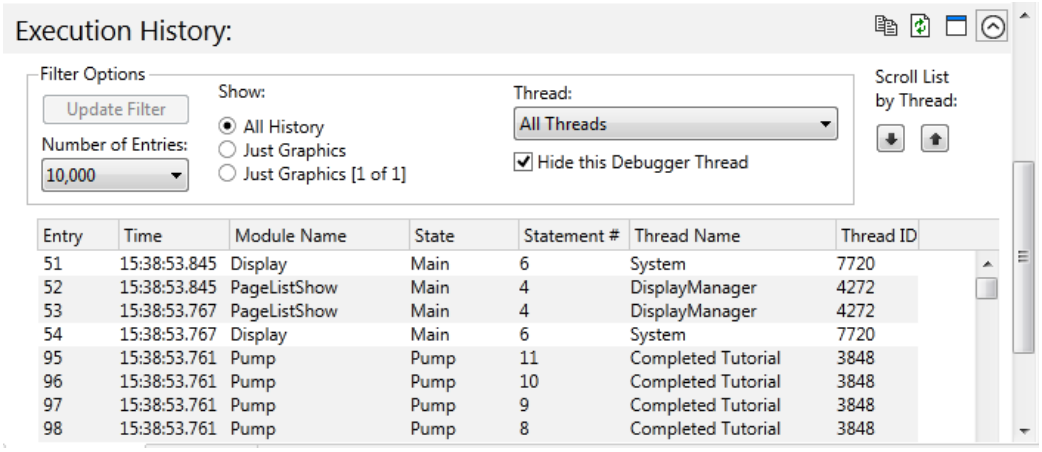
1. Enter a filtering condition in the Filter field in the module content window (e.g. to view all variables whose name ends with 1, you could enter *1. To view all variables whose name begins with "a" you could enter a*).
2. Press the Enter key.

The module content window will display only those objects that meet the entered filtering condition.

Note: To clear the filter, enter * in the Filter field and press the Enter key.

Working with the Execution History

The Execute History display of the System Summary panel shows a history of threads that have run, sorted so that the most recent are at the top of the display.

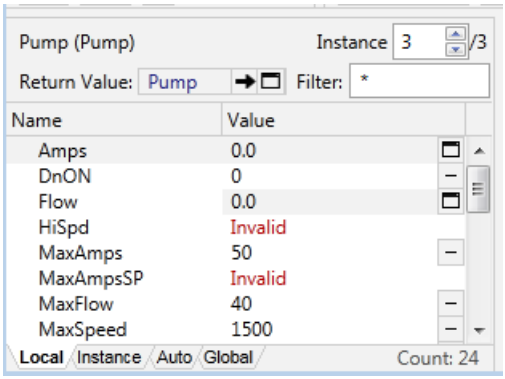


Note that, when used in the Summary (Live) view, the list does not update dynamically. You must click the Refresh button in order to see new threads. You can use the Copy button to copy the window contents to the Windows™ clipboard for review elsewhere.

The arrows labeled Scroll List by Thread allow you to step over entries in the list having the same thread ID, thereby scrolling quickly through the display to see each thread that was running.

Double-click on any entry in the list in order to a) open that module in the module content list window, and b) enable filtering of the display for modules that either call the selected module, or thread entries for modules matching the one selected.

For example, a double-click on the entry 95 in the preceding figure results in the following:



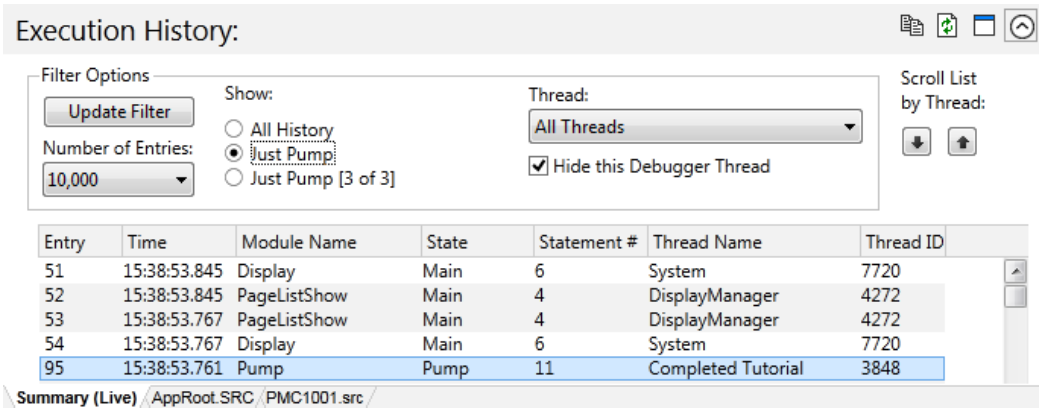
Related Information:

...Filter the Thread History - tools and examples.

...Select the Thread to Display - example.

Filter the Thread History

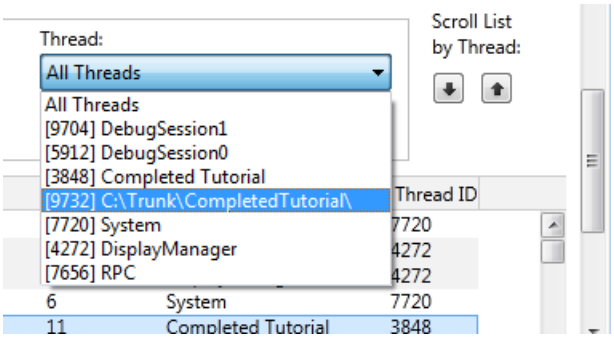
After double-clicking on an entry in the history list, you can filter the display for modules that call the one selected, or only for thread instances of the selected module.



By selecting the Show option., "Just Pump" and clicking the Update Filter button, you will limit the display to modules and threads related to module, Pump.
By selecting the Show option., "Just Pump [3 of 3]" and clicking the Update Filter button, you will limit the display to only threads belonging to the selected instance of Pump. The numbers in square brackets refer to the module instance and correspond to the instance selected in the module content list window:

Select the Thread to Display

Use the Thread selector to limit the display to show only instances of a given thread.



After selecting a thread to display, you must click the Update Filter button before the list will change.
Modules related to the Source Debugger are hidden from the thread history list by default.

Copying and Pasting Code Using the Source Debugger

The Source Debugger enables you to copy and paste any kind of object. You can cut or copy (using the Cut or Copy buttons or Ctrl + X and Ctrl + C respectively):

- When a module "node" is highlighted (i.e. has a blue background) in one of the three module trees (i.e. the static module tree, scope module tree, or call module tree).

- When a data value is highlighted in:
  - The main code display area
  - The watch window (except for expressions)
  - The pop-up data windows
  - The array view window

You can paste using Ctrl + V:

- When a data value is highlighted in any of the above.

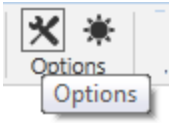
Note: If an object cannot be cut, copied, or pasted, the buttons for these commands will be disabled.

If you simply highlight a data quantity and paste, the value on the clipboard will be pasted into the highlighted variable, if the variable holds a type than can be pasted over.
The type of the value being pasted is also pasted so you can, for example, paste an object reference, image, array, stream, etc.
If you highlight and then single left-click a variable to enter edit mode, the value on the clipboard will be pasted into the edit field as a text representation of the copied quantity (e.g. the name of an object is pasted, rather than an object value). The text representation of the value is made available in text form, to applications outside the Source Debugger and to processes outside VTScada.

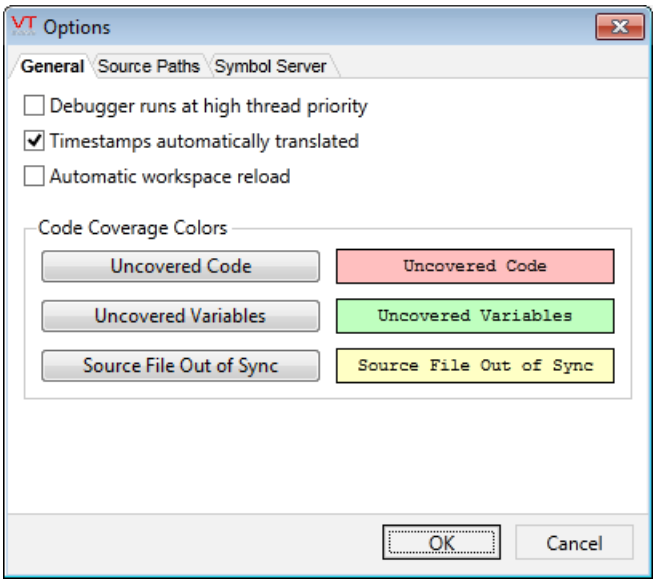
Note: Not all VTScada values can be represented as text. For those that cannot, the text "Valid" or "Invalid" will be pasted.

Source Debugger Options

The Source Debugger enables you to configure some basic options for the way it behaves. These options can be modified using the Options dialog.



To launch the Options dialog, click Options in the Source Debugger's tool bar. Example follows:



The sections that follow provide an overview of the options that can be set using each tab of the Options dialog.

Related Information:

...Source Debugger Options Dialog: General Tab – control of debugger thread priority, timestamps, workspace reload and code coverage colors.

...Source Debugger Options Dialog: Source Paths Tab – enables the Source Debugger to specify the path to the correct version of a source file when the source file is stored on a PC other than the one on which the Source Debugger is running.

...Source Debugger Options Dialog: Symbol Server Tab – ensure that the binary image contained in a dump file is interpreted using the correct version of VTScada.

Source Debugger Options Dialog: General Tab

Debugger runs at high thread priority

The Source Debugger is a VTScada script application, and as such is subject to equal time-sharing of the CPU with other VTScada applications. If this check box is selected, the proportion of time the Source Debugger gets on the CPU (with respect to other VTScada applications) will be increased. This has the effect of allowing the Source Debugger user

interface to respond more quickly when another VTScada application is consuming more CPU time than it should. It is useful to analyze the reason that another VTScada application is behaving in that way.

Timestamps automatically translated

Timestamps are represented in VTScada double-precision floating-point numbers (e.g. 1129884300.250). Checking the Timestamps automatically translated check box will cause doubles that are in the range for timestamps to be displayed in a more readable format. For example, 1129884300.250 will translate to Oct 21, 2005 08:45:00.25.

Automatic workspace reload

If checked, your workspace (selected module, breakpoints and watches) will automatically be saved while you work. When you stop the source debugger then re-open it, the automatically-saved workspace will be loaded immediately, allowing you to continue working from the point where you stopped.

Code coverage Colors

Three highlight colors are used when using the code coverage feature. If you find the default colors unsatisfactory, you may select others. Please see the section, Code Coverage for more information about what these highlight colors mean.

Related Information:

...Code Coverage

Source Debugger Options Dialog: Source Paths Tab

It is important that the Source Debugger always opens the correct source file (.SRC or .WEB file) and symbol file (.SYM) for the module being examined.

The Source Paths tab enables the Source Debugger to specify the path to the correct version of a source file when the source file is stored on a PC other than the one on which the Source Debugger is running. It further

enables the correct version of a source file to be opened when examining a dump file.
When there are no entries in the fields of the Source Paths tab, the Source Debugger attempts to open the file from which the .RUN file was compiled. For example, if C:\VTScada\VTS\AnalogIn.RUN is the .RUN file containing the module, "AnalogInput" and it was compiled from the source file C:\VTScada\VTSAnalogIn.SRC, the Source Debugger will attempt to open the source file C:\VTScada\VTS\AnalogIn.SRC and the symbol file C:\VTScada\VTS\AnalogIn.SYM.
A problem arises when the file C:\VTScada\VTS\AnalogIn.SRC is not the source file that was compiled to produce the .RUN file. A simple solution is to simply specify a list of paths to be searched for the source/symbol files. This, however, is not ideal, as it is possible for a module in an application layer to have the same name as one in another layer (e.g. the VTScada layer). Therefore, the specification of search paths is more complex. Thus there are three elements to the Source Paths tab of the Options dialog that control the source file searching behavior.

Substitute VTScada Installation Path

The Substitute VTScada Installation Path field will accept a path to an alternative VTScada installation folder tree. Any module compiled from a source file located within the VTScada installation folder tree will have their source/symbol files loaded from the path specified in this field.

External Folders

The External Folders list will accept one or more paths outside the VTScada installation folder tree. Any module compiled from a source file outside the VTScada installation folder tree will have their source/symbol files loaded from a path in the External Folders list. The list is parsed from top to bottom (i.e. if the source/symbol files cannot be found in the path at the top of the list, the next path down the list will be searched). Searching stops when a file is found or the end of the list has been reached.

Always Search

The Source Debugger normally only searches the paths specified in the Substitute VTScada Installation Path field and the External Folders list if the source file indicated by the module is not found. If the Always Search check box is selected, the Source Debugger will be prevented from search for the source/symbol files in the path indicated by the .RUN file. Instead the Source Debugger will be forced to search for the source/symbol files in the paths specified in the Substitute VTScada Installation Path field and the External Folders list.

The following will assist you in the proper configuration of the Source Paths tab when you are analyzing a dump file.

Analyzing a Dump File

Most of the time, the version of VTScada that is contained in a dump file will not match the version of VTScada that is running the Source Debugger. In such cases, you want the Source Debugger to search a specific location for the source/symbol files, rather than searching the folder in which the version of VTScada running the Source Debugger is located.

For files within the dumped VTScada installation folder tree:

1. Create a folder tree that mimics the dumped VTScada installation folder tree.
2. Copy the source and symbol files for the version of VTScada corresponding to the dumped version.
3. Paste the copied files into the new folder tree created in step 1.
4. Enter the path of the new tree into the Substitute VTScada Installation Path field.
5. Select the Always Search check box.

For files outside the dumped VTScada installation folder tree:

1. Create one or more folders.
2. Copy the source and symbol files for the version of VTScada corresponding to the dumped version.
3. Paste the copied files into the new folder tree created in step 1.
4. Enter the path(s) of the new folders into the External Folder list, using the Add button.

5. Use the Move Selected buttons to arrange the folder list correctly.
6. Select the Always Search check box.

Remote Debugging Using the VIC

For files within the dumped VTScada installation folder tree:

1. Create a folder tree that mimics the dumped VTScada installation folder tree.
2. Copy the source and symbol files for the version of VTScada corresponding to the dumped version.
3. Paste the copied files into the new folder tree created in step 1.
4. Enter the path of the new tree into the Substitute VTScada Installation Path field.
5. Deselect the Always Search check box.

For files outside the dumped VTScada installation folder tree:

1. Create one or more folders.
2. Copy the source and symbol files for the version of VTScada corresponding to the dumped version.
3. Paste the copied files into the new folder tree created in step 1.
4. Enter the path(s) of the new folders into the External Folder list, using the Add button.
5. Use the Move Selected buttons to arrange the folder list correctly.
6. Deselect the Always Search check box.

The source files will be loaded from the server to which the VIC is connected, unless they are not found, in which case the files will be searched for on the PC running the VIC. The paths specified must therefore be on the PC running the VIC. This enables you to remotely debug a production system that does not have the source code installed on it, or has only a subset of the source code (e.g. an application layer).

Local Debugging of a Production System

For files within the dumped VTScada installation folder tree:

1. Create a folder tree that mimics the dumped VTScada installation folder tree.
2. Copy the source and symbol files for the version of VTScada corresponding to the dumped version.

3. Paste the copied files into the new folder tree created in step 1.
4. Enter the path of the new tree into the Substitute VTScada Installation Path field.
5. Deselect the Always Search check box.

For files outside the dumped VTScada installation folder tree:

1. Create one or more folders.
2. Copy the source and symbol files for the version of VTScada corresponding to the dumped version.
3. Paste the copied files into the new folder tree created in step 1.
4. Enter the path(s) of the new folders into the External Folder list, using the Add button.
5. Use the Move Selected buttons to arrange the folder list correctly.
6. Deselect the Always Search check box.

By using paths that are either UNC names or shares on mapped drives, source files will be loaded from the production system unless they are not found, in which case they will be loaded from a networked file system.

This enables you to debug a production machine live on site, while not having to copy source files onto it.

Debugging a Production System Using Remote Control

For files within the dumped VTScada installation folder tree:

1. Create a folder tree that mimics the dumped VTScada installation folder tree on the production system.
2. Copy the source and symbol files for the version of VTScada corresponding to the dumped version.
3. Paste the copied files into the new folder tree created in step 1.
4. Enter the path of the new tree into the Substitute VTScada Installation Path field.
5. Deselect the Always Search check box.

For files outside the dumped VTScada installation folder tree:

1. Create one or more folders on the production system.
2. Copy the source and symbol files for the version of VTScada corresponding to the dumped version.
3. Paste the copied files into the new folder tree created in step 1.
4. Enter the path(s) of the new folders into the External Folder list, using the Add button.
5. Use the Move Selected buttons to arrange the folder list correctly.
6. Deselect the Always Search check box.

As indicated in the instructions above, you will have to copy the source/symbol files onto the production system, however, you may copy the source/symbol files into separate folder trees on the production system, making tidy-up at the end of the debugging session easier by allowing you to later delete the source/symbol files from the separate tree. This minimizes the risk of your inadvertently deleting important files from the production system when debugging using source files that you do not want to leave on the production system when finished.

Note: The user interface of your debugging session will be running on the production server's display. Ensure that this does not interfere with system operation. While this method of debugging can be used, other methods are preferred.

Source Debugger Options Dialog: Symbol Server Tab

In order to interpret the binary image that a dump file contains, the Source Debugger requires information about the internal structure of the VTScada engine (VTS.exe). As this changes from version to version of the engine, the Source Debugger uses a program database (.PDB) that is generated as part of the process of building VTS.exe.

How Does the Source Debugger Locate the Correct .PDB File?

The Symbol Server tab enables the Source Debugger to locate the correct .PDB file for the dump image being examined.

Note: The default setting for the Local Symbol Cache Path field (which is pointed to the DumpTools subdirectory within the VTScada installation directory (e.g. C:\VTScada\DumpTools) where the .PDB file is stored) should result in the correct .PDB file being located by the Source Debugger. The Symbol Server Location field is for internal use by Trihedral only, and as such does not require input.

(Trihedral only) The Source Debugger uses a two-phase process to locate the correct .PDB file:

- The Local Symbol Cache Path is examined for the .PDB file.
- If the correct .PDB file is not located in the DumpTools directory, the Source Debugger will search a symbol server (Microsoft terminology) for the correct .PDB file.
- A symbol server can hold many versions of .PDB (one for each version of VTScada that is released). Internally, this is normally a network share (e.g. \\<trihedralserver>\VTSSymbols), but can be a URL (e.g. http://<trihedralservername>.trihedral.com/VTSSymbols).

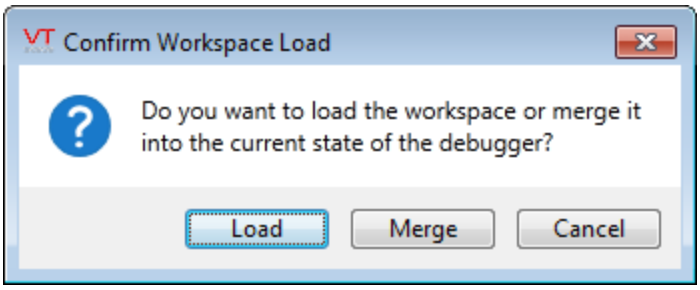
Both fields do not have to be filled in. If only the Local Symbol Cache Path field is filled in (as it is by default), the .PDB file is expected to be found there. If both fields are completed, the operation is as described as above, with PDBs fetched from the symbol server stored in the Local Symbol Cache Path, there by reducing network traffic for the remainder of this debugging session and for future sessions that require the same .PDB. The .PDB files cached locally can be manually deleted if no longer required.

If only the Symbol Server Location field is filled in, there is no local caching of the .PDB file, and all accesses to the desired .PDB file are done over the network. This is a slower process, but avoids storing a .PDB file on the local hard disk.

Confirm Workspace Load

A Workspace is the set of the selected module, the breakpoints, and the watches in your session. You may save a workspace explicitly by using

the Save Workspace button in the toolbar, or you may set the Auto Workspace Reload check box in the Options menu.



When loading a workspace, either by using the Open Workspace button to load a selected workspace file that you saved earlier, or by using the Reload Auto Workspace button to load the last automatically saved workspace, you may choose to either replace the workspace you are working in, or you may merge the current workspace with the one you are opening.

Code Coverage

The code coverage feature is used to see what modules, states, statements and variables have been executed or used. This is useful in a number of ways, not least of which are:

- Code that has not run cannot have caused an error.
- Code that should have run but hasn't, may indicate the location of an error.

Related Information:

...The Code Coverage Display - Shows the module tree, where each module has a colored indicator showing what percentage of the module has been covered by running code.

...Refreshing the Code Coverage Display - how to refresh the display.

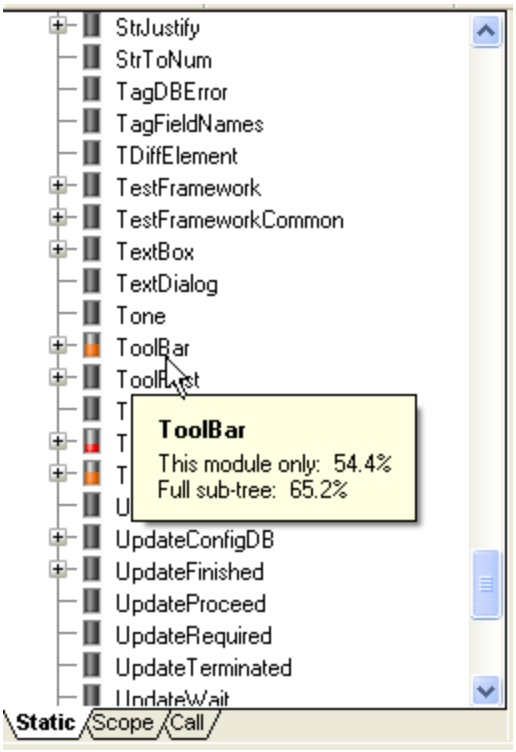
...Stepping Between Blocks of Covered Code - instructions for dealing with large gaps.

...Using a Code Coverage Merge File - how and why the current record may be saved.

... Resetting the code coverage counts - how to obtain a clear view of code coverage from a fresh starting point.

The Code Coverage Display

When code coverage mode has been switched on, using the Toggle Code Coverage button, the module tree and the code display windows will change. The module tree will display each module with a colored indicator that shows what percentage of the module has been covered by running code.

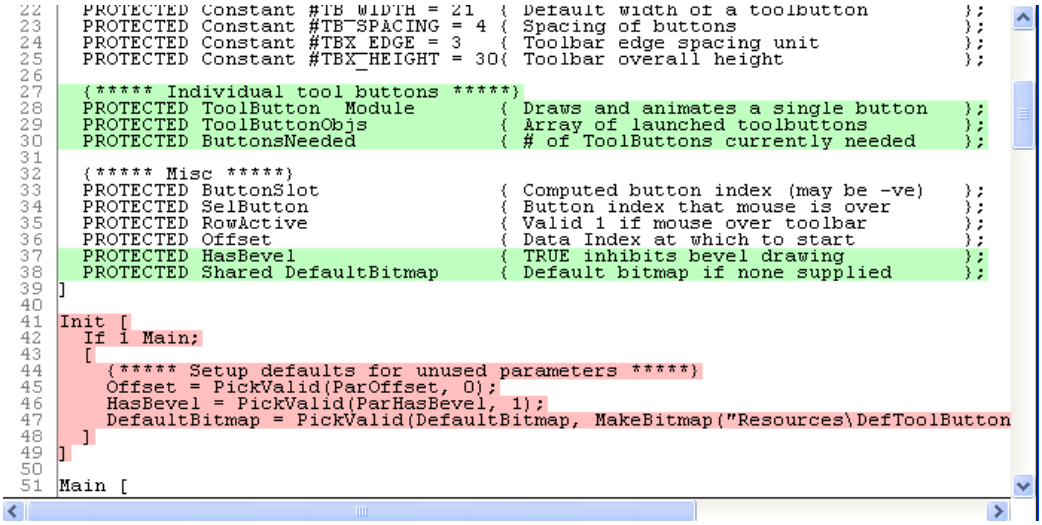


Resting the mouse pointer over a module causes a tool tip to appear, showing the code coverage within the module, and within the entire module sub-tree.

The color indicators are as follows:

- Grey – no code coverage.
- Red – minimal code coverage.
- Orange – medium coverage.
- Yellow – high coverage.
- Green – full coverage.

The code display window will look similar to the following:



The highlight colors are user-configurable. (See: Source Debugger Options). The default colors have the following meanings:
Green background: Indicates variables that have not been covered.
Red background: Indicates statements that have not been covered.
White lines indicate code that has been run and variables that have been used.

Note: If the entire window changes to a yellow background, this indicates that the source code has been changed since the application was last compiled. For the code coverage system to work, all changes must be compiled. You may need to re-start the application.

Refreshing the Code Coverage Display

The code coverage counter is always running, but the display does not update dynamically. To update the display you should use the Refresh Code Coverage Information button in the toolbar. This will provide you with an up-to-date view of what statements and variables have been used.

Stepping Between Blocks of Covered Code

In a module with minimal code coverage, there may be long gaps between the uncovered highlights. The Next Highlight and Previous Highlight buttons allow you to quickly jump from one block of uncovered code to the next.

Using a Code Coverage Merge File

You can save the current record of what code has been covered to a file, and then merge that information back to the current display. This can be useful in at least two situations:

- If you are working as part of a team, each team member can test a different part of a large application and save their code coverage history to a file. These files can then be collected together and merged into one display to provide a complete picture of what code and variables have been used in an application
- You may wish to reset the counters to zero before running a test, but not want to lose your current code coverage information. You can save the current code coverage information to a file before performing the reset, then, once you have finished your test, merge the code coverage information back into the current display.

Resetting the code coverage counts

To reset all the counters to zero for a fresh start on the code coverage, use the Reset button in the toolbar. Use this to obtain a clear view of code coverage from a fresh starting point.

Test Framework Application

The Test Framework application enables VTScada programmers to test VTScada applications. This application can be launched from the Source Debugger or from the VAM.
Tests are organized on an application-by-application basis. You may choose which application to test, and Test Framework will load an independent copy of the application and automatically discover all tests that are defined within the application.

Note: It is the responsibility of the VTScada programmer to write their own tests for their individual applications. Information on writing tests for the Test Framework is provided in Writing Tests for the Test

Framework.
The Framework may be used by customers for their own modules, but no technical support is provided. Please request the Test Framework package, available from the Trihedral Marketing department.

A tree showing all tests and all modules that directly or indirectly contain tests will automatically be generated. Tests may be enabled or disabled on an individual basis, or at a tree-node level.
For an understanding of how to design and use tests, the following Internet sites may be of use:

...http://www.agiledata.org/essays/tdd.html

...http://en.wikipedia.org/wiki/Test-driven_development

Related Information:

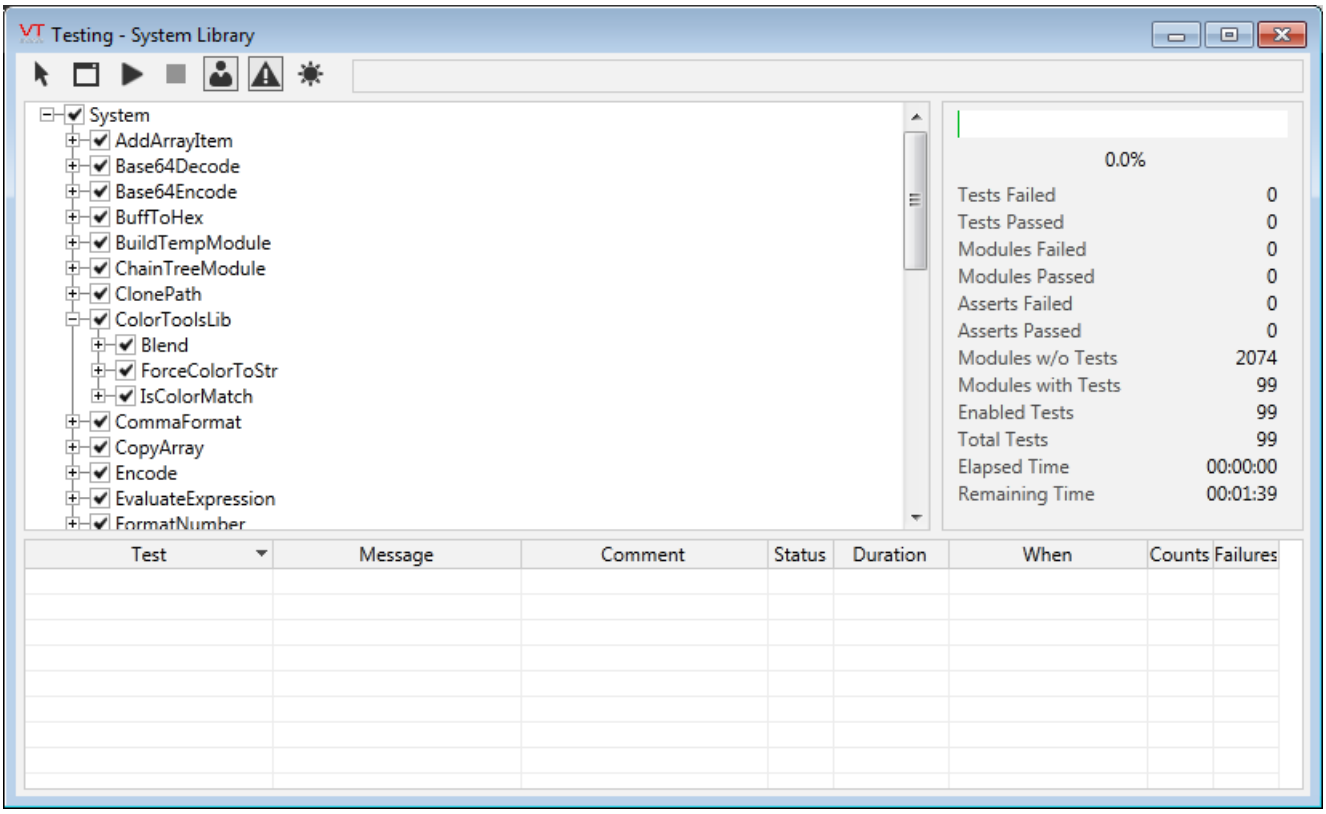
...Test Framework Application Components - description of the user interface components.

...Writing Tests for the Test Framework - tools and recommended methods.

...Running Tests - how to run tests and select which results to view.

Test Framework Application Components

An example of the Test Framework application:



Components:

Select Application Window

The Select Application Window button (first button from the left) enables you to select a running application for testing (see Selecting an Application for Testing Using the Test Framework).

Select Application from List

The select application from list button (second button from the left) launches the Select Application dialog that displays a list of VTScada applications that can be selected for testing.

Message Area

To the right of the tool bar buttons appears the message area field. Any important information about the application selected for testing will be displayed in this field (e.g. application needs to be compiled).

Module Tree

The module tree area appears just under the toolbar to the left of the dialog, and displays the modules that are included in the current test. You may use the module tree to select/deselect the tests you wish to run within the loaded application (see Running a Test Using the Test Framework).

Summary Area

The summary area displays a summary of the testing, including statistics on:

Tests Failed	The number of tests that did not pass during this run.
Tests Passed	The number of tests that passed during this run.
Modules Failed	The number of modules that failed during this run.
Modules Passed	The number of modules that passed during this run.
Asserts Failed	The number of Asserts that failed during this run.
Asserts Passed	The number of Asserts that passed during this run.
Modules w/o Tests	The number of modules that did not include tests.
Modules with Tests	The number of modules that included tests.
Enabled Tests	The number of tests that were enabled.
Total Tests	The total number of tests that were run.
Elapsed Time	The total elapsed time (in hh:mm:ss) that it took to run the test.
Remaining Time	The total amount of time (in hh:mm:ss) outstanding for the test being run.

Run Tests

The Run Tests button can be clicked to run tests on the selected modules.

Pause Tests

The Pause Tests button can be clicked to pause the tests that are running.

Resume Tests

The Resume Tests button can be clicked to resume testing once a test has been paused.

Stop Tests

The Stop Tests button can be clicked to stop the running tests.

Suppress Manual

Check this box to select that only the fully-automated portions of the tests should run. This depends on the test being written to support the flag.

Failures Only

The Failures Only radio button can be clicked if you wish the test list to display information only on those tests that failed .

Show All

The Show All radio button can be clicked if you wish the test list to display information on all tests, passed and failed.

Test List

The test list presents information on the tests that have passed or failed (depending upon whether the Failures Only or the Show All radio button has been selected). Tests that passed are presented in green text, while tests that failed are highlighted in red. The test list presents the following information for each test:

Test	The Test column identifies the name of the application directory, source file, and module for the test.
Message	The Message column presents a short message identifying the value that was expected and the value that was returned on completion of the test.
Comment	The Comment column displays any comments relevant to the test.
Status	The Status column indicates whether the test passed or failed.
Duration	The Duration column displays the amount of time (in milliseconds) that it

	took for the test to complete.
When	The When column indicates the exact time and date that the test was completed.
Counts	The Counts column indicates the number of times the test was run. (This count is incremented each time the Run Test button is clicked.)
Failures	The Failures column indicates the number of times the test failed within each testing session.

Related Information:

...Writing Tests for the Test Framework

...Running Tests

...Viewing Test Results

Writing Tests for the Test Framework

Tests are defined by creating a module of class Test within the module to be tested. The following guidelines should be observed:

- A module may contain zero or more tests.
- Tests may not contain other tests.
- Each test must either return Invalid if it is a subroutine, or slay itself upon completion.
- A test must call one or more Assert functions.
- Tests are passed two parameters:
  Owner: Module value of the module to test (static parent of the test module)
  Framework: Object value of the Test Framework code where the Assert functions are defined.
- A test is responsible for creating the environment for the module under test (MUT). This can be done by defining globally-referenced variables and modules in the MUT within the Test module itself, or with the use of Fixtures, or both.
- Variables outside the scope of the MUT must exist in the Test module or in a fixture.
- A test is run in its own thread.

- The MUT is reloaded for each test to make sure all temporary variables are cleared before each test
- Tests run with no parent scope aside from any fixtures declared.

Related Information:

...Assert Subroutines – the eight available subroutines.

...Fixture Modules – creation and use of.

...Using the ThreadIdle Function – how to use in the test framework.

Related Functions:

... SetOverride – used to override op codes with a specified script.

Assert Subroutines

Asserts are subroutines that are called within a test to validate the results of the test. Ideally, there should be one assert per test.
Eight Assert functions are defined:

- AssertTRUE(Condition, Comment)
- AssertFALSE(Condition, Comment)
- AssertEqual(Expected_Value, Actual_Value, Comment, Tolerance)
- AssertNotEqual(Expected_Value, Actual_Value, Comment)
- AssertGreater(Expected_Value, Actual_Value, Comment)
- AssertGreaterEqual(Expected_Value, Actual_Value, Comment)
- AssertLess(Expected_Value, Actual_Value, Comment)
- AssertLessEqual(Expected_Value, Actual_Value, Comment)

The parameters are:

Condition	Boolean condition determining pass/fail state of the test. Invalid is considered a failure
Comment	Brief comment to help identify the source of a failure
Expected_Value	Typically a constant that defines what the expected value should have been
Actual_Value	Value observed during the test
Tolerance	Maximum allowable difference between Expected_Value and Actual_Value for equivalence test of numbers. Defaults to 0.

If the condition asserted by the call is true, then the Assert has passed and no failure will be recorded. All Asserts for a test must pass in order for the test to pass.

Fixture Modules

Fixtures are a mechanism used to handle common test environments. A fixture is created by defining a module with no special class in a scope higher than the tests using the fixture. Fixtures are launched in the order declared within the test. They become the scope tree for the Test module.
To have a test use a fixture, you must define a variable of type FIXTURE with the same name as the fixture module within the test. The fixtures are launched in the order declared within the test.
Fixtures may optionally contain a FixtureReady variables. If this variable is defined, the Test Framework will wait until this variable becomes TRUE before launching the next fixture or the test itself.

Using the ThreadIdle Function

A ThreadIdle function has been created to help support testing. It takes a single parameter which is an object value implying a thread. The return value is TRUE when there are no statements on the to do list for the thread. It does not consider timers in the return value.
This function is useful when instantiating the MUT and waiting for the initialization of the MUT to complete before executing the Assert calls.

Related Functions:

... ThreadIdle

Running Tests

Note: Your application must have tests written especially for it if you wish to use the Test Framework application to test it. Please read Writing Tests for the Test Framework to learn about writing tests for your application.

Selecting an Application for Testing Using the Test Framework

There are two means by which a VTScada application can be selected for testing using the Test Framework application.

A) Selecting a Stopped Application for Testing

1. Run the Test Framework application.
2. Click the Select Application from List button (second button from the left in the tool bar). The Select Application dialog will open.
3. Select the VTScada application containing your tests.
4. Click OK. The Select Application dialog will close, and the selected application will load into the module tree of the Test Framework application.

B) Selecting a Running Application for Testing

1. Run the Test Framework application.
2. Run the VTScada application you wish to test using the VAM.
3. Click the Select Application Window button (first button from the left in the tool bar).
4. Click anywhere on the running VTScada application containing your tests. The selected application will load into the module tree of the Test Framework application.

Once the application has been loaded into the Test Framework application, you may run a test.

Related Tasks:

...Running a Test - how to run, pause, and resume tests.

...Viewing Test Results -selecting the result set to view.

Running a Test

Once you have selected an application for testing using the Test Framework (see Selecting an Application for Testing Using the Test Framework), you may run a test.

1. Select the modules containing the tests you wish to run for the loaded application.
   To do so, click the plus button to the left of each module to expand the module tree. Any modules you wish to test should display a checkmark to their

left. If you wish to test all modules, ensure that the root module at the top of the tree displays a checkmark.

2. Click the Run Tests button. The progress of the tests will be indicated by the progress bar just above and to the left of the Run Tests button. Once the tests are complete, the test list should populate with data.
3. Select the test data you wish to review:
4. Select the Show All radio button to review data on all tests, both passed and failed, OR
5. Select the Failures Only radio button to review data on only those tests that failed.

Pausing a Test Using the Test Framework

If you wish to pause a running test, click the Pause Tests button.

Resuming a (Paused) Test Using the Test Framework

If you have paused a test and wish to continue running it, click the Resume Tests button.

Stopping a Test Using the Test Framework

If you wish to stop a running test, click the Stop Tests button.

Viewing Test Results

Viewing All Tests (Passed and Failed) Using the Test Framework

To view data on all tests, both passed and failed, select the Show All radio button. The test list will display any passed tests in green text, while any failed tests will be highlighted in red.

Viewing Failed Tests Using the Test Framework

To view data on only those tests that have failed, select the Failures Only radio button. The test list will display any failed tests highlighted in red. If no tests have failed and the Failures Only radio button has been selected, the test list will appear empty.

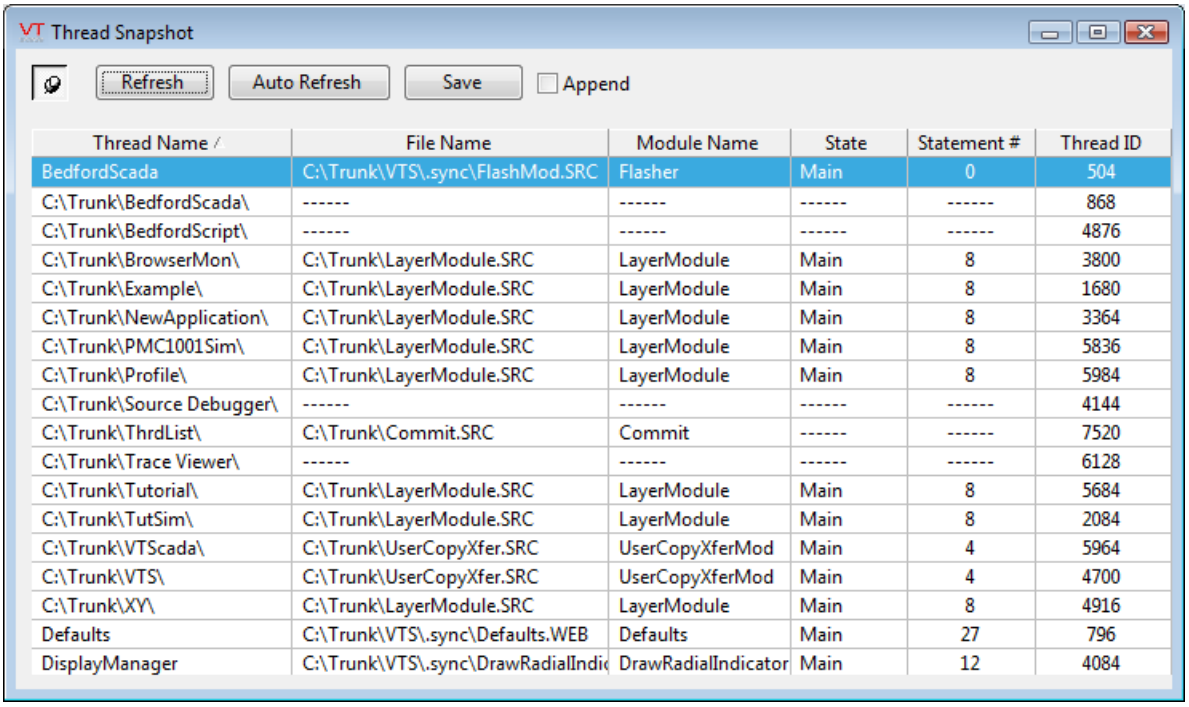
Thread List Application

Applications, as well as some system components, run in their own execution threads so that they do not interfere with the execution of other applications or components. The Thread List analysis utility provides a list of the separate threads of execution for which VTScada is responsible.

Note: If the Thread List application is not referenced in the VAM, you must manually add it. The Thread List application's directory is named "ThrdList", and is stored within the VTScada installation directory.

If an execution thread becomes blocked (i.e. freezes or gets stuck while executing a statement), the Thread List analysis tool can be used to identify the offending statement that is causing the blockage. This information provides a starting point for examining the scripting code that has caused the thread to be blocked.

The Thread List utility:



Thread Name	File Name	Module Name	State	Statement #	Thread ID
BedfordScada	C:\Trunk\VTS\.sync\FlashMod.SRC	Flasher	Main	0	504
C:\Trunk\BedfordScada\	------	------	------	------	868
C:\Trunk\BedfordScript\	------	------	------	------	4876
C:\Trunk\BrowserMon\	C:\Trunk\LayerModule.SRC	LayerModule	Main	8	3800
C:\Trunk\Example\	C:\Trunk\LayerModule.SRC	LayerModule	Main	8	1680
C:\Trunk\NewApplication\	C:\Trunk\LayerModule.SRC	LayerModule	Main	8	3364
C:\Trunk\PMC1001Sim\	C:\Trunk\LayerModule.SRC	LayerModule	Main	8	5836
C:\Trunk\Profile\	C:\Trunk\LayerModule.SRC	LayerModule	Main	8	5984
C:\Trunk\Source Debugger\	------	------	------	------	4144
C:\Trunk\ThrdList\	C:\Trunk\Commit.SRC	Commit	------	------	7520
C:\Trunk\Trace Viewer\	------	------	------	------	6128
C:\Trunk\Tutorial\	C:\Trunk\LayerModule.SRC	LayerModule	Main	8	5684
C:\Trunk\TutSim\	C:\Trunk\LayerModule.SRC	LayerModule	Main	8	2084
C:\Trunk\VTScada\	C:\Trunk\UserCopyXfer.SRC	UserCopyXferMod	Main	4	5964
C:\Trunk\VTS\	C:\Trunk\UserCopyXfer.SRC	UserCopyXferMod	Main	4	4700
C:\Trunk\XY\	C:\Trunk\LayerModule.SRC	LayerModule	Main	8	4916
Defaults	C:\Trunk\VTS\.sync\Defaults.WEB	Defaults	Main	27	796
DisplayManager	C:\Trunk\VTS\.sync\DrawRadialIndi	DrawRadialIndicator	Main	12	4084

As displayed above, the Thread List is composed of the following elements:

Pin The pin button can be selected to always keep the Thread List window on top, or can be deselected to allow the Thread List window to be minimized.
Refresh The Refresh button can be clicked to update the display of threads.
Auto Refresh The Auto Refresh button can be latched on to result in the display of threads being updated automatically twice a second.
Save The Save button can be clicked to save the current thread list data to either a text file or a .CSV file. A file dialog will prompt for a name and provide a choice of setting the extension to either ".txt" or ".csv". In older versions of VTScada, this file name would have been "ThreadLog.txt", located within the ThrdList directory which is located within the VTScada installation directory (e.g. C:\VTScada\ThrdList\ThreadLog.txt).
Append The append button can be selected to add new data to existing files rather than over-writing them.
Thread Name The Thread Name column displays the name or title assigned to each thread.
File Name The File Name column displays the path to and name of the document file that contains the executing statement.
Module Name The Module Name column displays the name of the module within the document file (specified in the File Name column) that contains the executing statement.
State The State column displays the state within the module that contains the executing statement.
Statement # The Statement # column displays the numerical index of the executing statement within the state.
Thread ID The Thread ID column displays the ID number for each thread.

Trace Viewer Application

Note: The Trace Viewer application should not be confused with the Trace VTScada Actions application, which enables you to select

different VTScada services (such as the RPC Manager and Modem Manager), and actions (the Navigator or SQL calls), and monitor the selected items by saving pertinent data about them (such as the date and time they executed) to disk.

The Trace Viewer application enables you to monitor the content and parameters of driver messages and VTS-related network traffic as it occurs. The collected data is also logged to standard Access database files for later viewing.

There are three components involved in tracing:

- The communications to be traced. Each message source, (driver tracing, RPC diagnostics and SOAP message tracing) has its own format and therefore is stored in its own database.
- The module that collects and writes the actual data to the database. This is DBTrace, a system-level module that maintains live communications with the Trace Viewer.
- The viewer. This is the Trace Viewer application, where you can select, filter and display the communications data.

Note: Users of older versions of VTS may be familiar with the RPC Diagnostics application. That application is now obsolete. The Trace Viewer application enables you to view RPC trace information in real-time.

Trace files are regular Microsoft Access .MDB files. These are stored in the TraceFiles subdirectory within the VTScada installation directory. These files are deleted as they age, with the default set to 30 days. If you wish to modify this setting, you may do so using the DbTraceDaysToPreserve variable in Setup.ini. You may also control the size using .DBTraceFileSize.

Trace files can contain additional tables that assist in the interpretation and filtering of their data. This means that the files can be exported into another environment and can still be interpreted correctly.

Related Information:

...Trace VTScada Actions Application

What the Trace Viewer can show you

Depending on the license you purchased with VTScada, up to five sources of information are available to the trace viewer.
Note: Service names that contain a child-tag delimiter will be shown with a forward-slash, rather than the more usual back-slash.

Driver Messages for running applications (always available while an application is running)

Information collected includes:

- Timestamp, accurate to the nearest thousandth of a second
- Direction (Sent or Received from the driver)
- Service name
- Driver name as entered in the tag
- Driver area as entered in the tag
- Driver description as entered in the tag
- Port name that the driver is attached to
- Data included in the communication to/from the driver (a string of hexa-decimal values)

Historian Diagnostics (always available)

- Error messages only. One or more Historians must be selected for tracing. Information collected includes:
- Timestamp, accurate to the nearest thousandth of a second
- Historian name
- Trace type
- Tag name – the source of the data that did not write.
- Error code – a numeric code identifying the error.
- Error text.
- Message

RPC Diagnostics (always available)

Information collected for remote procedure calls includes:

- Timestamp, accurate to the nearest thousandth of a second
- Identification of internal events
- The message's routing flag
- Sequence number of the message
- Direction (To or From the server identified by the IP address)
- IP address of the message source or destination
- Name of the application sending or receiving the message
- Name of the service or machine
- Data and parameters of the message

SOAP Messages (available only if you have a license for VTScada Web Services)

Information collected includes:

- Timestamp, accurate to the nearest thousandth of a second
- Message number (large messages might be split across more than one line)
- Message status (an HTML error code indicating success or failure)
- Source IP address and Port number
- Destination IP address and Port number
- Indication of whether the message is incoming or outgoing.
- The SOAP action (the function call being made)
- Message size measured in bytes.
- The SOAP-encoded XML message

OPC Server Trace Messages (available only if you have a license for the VTScada OPC Server and a running application is using that server)

Information collected includes:

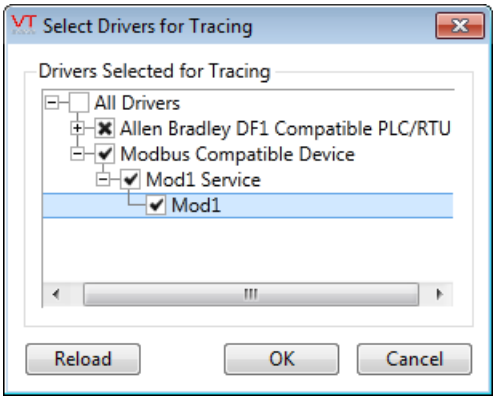
- Timestamp, accurate to the nearest thousandth of a second
- Event description
- Textual OPC item ID being read from or written to. (Often includes the tag name.)
- Numeric ID of the OPC property
- Write Value – the value being written to the tag

- Result – depending on the nature of the read or write operation, may be one or more of the following:
- the value being read or written,
- the quality of the data transfer
- the number of child nodes
- the name of the tag
- access rights
- type
- timestamp
- Items such as quality, access rights, type, etc are numeric codes. See: Properties of Tag OPC Items.

Features for Driver Tracing

You can choose which drivers are visible in the Trace Viewer by using the Select Drivers for Tracing dialog. This dialog displays a tree view of all the drivers used in the application. Select either all, or only those that are of interest to you.

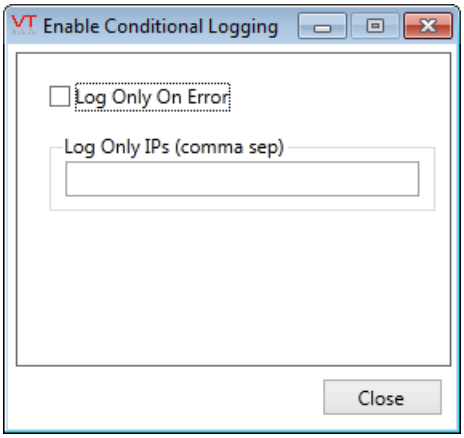
To open the dialog, click on the Select Drivers for Tracing Button. 
Note that this button is visible only when viewing the application's driver trace data source.
The dialog will look similar to the following:



Features for SOAP Message Tracing

SOAP message logging can generate large log files very quickly. For this reason, an option is provided to allow you to filter the messages that are

being logged.



The Enable Conditional Logging dialog provides two options:

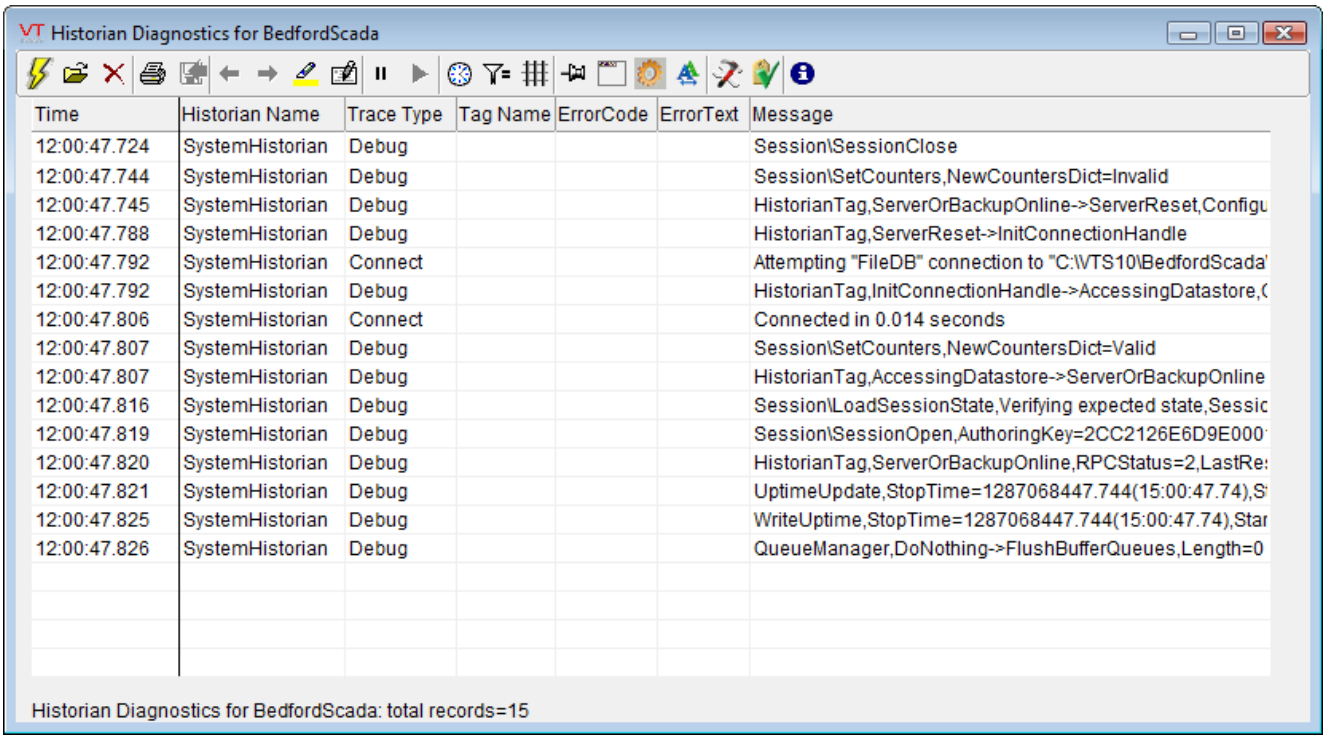
- You may opt to log only errors
- You may enter a comma separated list of IP addresses. Only messages coming from the specified addresses will be included in the log.

This dialog is displayed by selecting the Enable Conditional Logging button,  from the toolbar.

Features for Historian Diagnostics

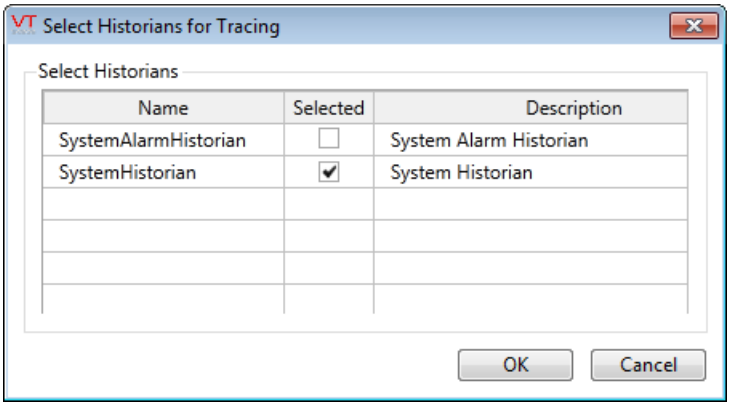
The Historian will log diagnostic messages upon startup. Following startup , diagnostic messages for the Trace Viewer will be logged only when errors occur. If neither of these conditions occur while you are capturing a trace, the display will be empty. An empty list is to be taken as a sign that all is working properly.

The following example shows typical startup messages.



You must select which historian's data to capture. When you first open the page, none will be selected by default. Click on the Select Historians Button.

The Select Historians dialog will present a list of all Historians active in running applications. Choose one or more by clicking on the Selected check box.



Related Information:

...Historian Trace Information – lists the details found in error messages.

...Historian Trace Options – select the trace sources to use.

Historian Trace Information

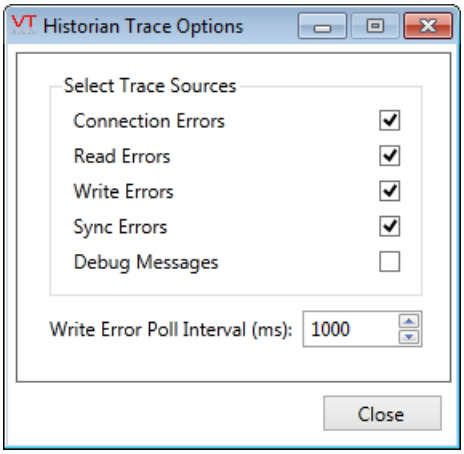
The following information about Historian errors will be displayed in the Trace Viewer:

- The time of the error.
- The name of the Historian tag
- The type of trace (debug or connection)
- The name of the tag whose data was being logged by the Historian
- An error code (of use only to Trihedral staff).
- The text of the error.
- A message describing the logged event.

Historian Trace Options

You may choose to enable or disable the display of the following sources of Historian Trace information:
Button to open the dialog:

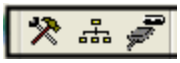
The Trace Options dialog:



This dialog also contains a Write Poll Interval field, measured in milliseconds. You may adjust this value to change the frequency with which trace information is gathered.

Features for Remote Procedure Call (RPC) Tracing

When RPC Diagnostics have been selected as the live capture data source, the toolbar will change to show three new buttons



 RPC Diagnostics – Settings

The RPC Diagnostics – Settings button is used to launch a dialog that enables users to set parameters that will affect the behavior of RPC traces.

See also: RPC Diagnostics Settings Dialog

 RPC Diagnostics – Services

The RPC Diagnostics – Services button can be clicked to launch the RPC Diagnostics – Services dialog that displays the current services and their state.

See also: Services Dialog

 RPC Diagnostics – Sockets

The RPC Diagnostics – Sockets button can be clicked to launch the Inter-machine Sockets dialog that displays the MachineNodes and their subordinate SocketNodes.

Related Information:

...Interpreting RPC Diagnostics Data – translations and filters for RPC Diagnostics.

...RPC Diagnostics Settings Dialog – reference: contents of this dialog.

...Inter-machine Sockets Dialog – reference: contents of this dialog.

...Inter-machine Sockets Data for Remote Machines – data displayed for each remote connection.

...Inter-machine Sockets Data for the Local Machine - data displayed for each local connection.

...Services Dialog - reference: contents of this dialog.

...Information Displayed for a Local Machine - possible machine states.

...Information Displayed for a Remote Machine - possible machine states.

...Information Displayed for a Client - reference: information displayed.

Interpreting RPC Diagnostics Data

The Trace Viewer performs the following translations and filters for RPC Diagnostics:

- IPs are translated to names where possible
- GUIDs are translated to application names
- Filtering can be done by IP/name, AppGUID, Service, trace point in RPC, or data content.

RPC Diagnostics Settings Dialog

The RPC Diagnostics Settings dialog is launched when the RPC Diagnostics –Settings button is clicked in the Trace Viewer's tool bar. The RPC Diagnostics Settings dialog enables you to set the following parameters:

Detailed Trace:
Enables the display of routing between the local workstation's RPC serialization FIFO and the transmission FIFOs, providing more detailed information on the internal operation of the RPC Manager. (The typical user does not require such information.)

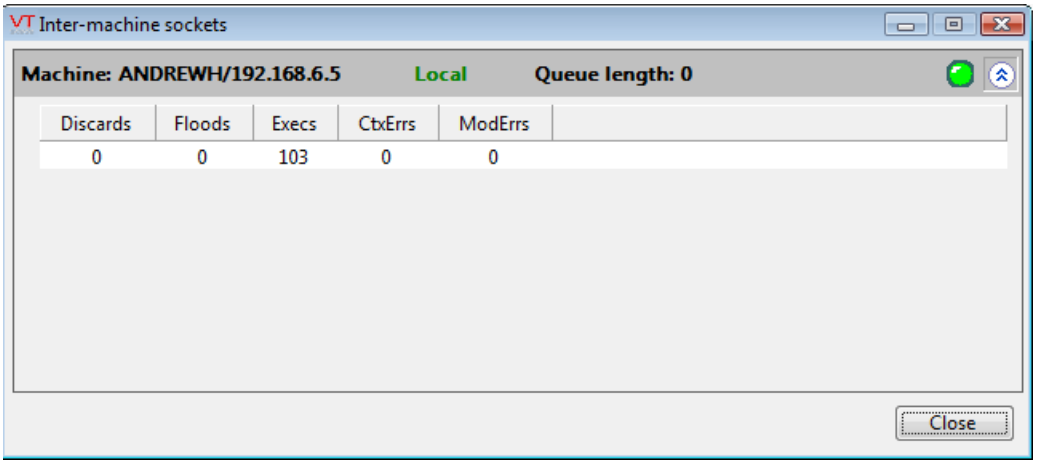
Max String Length:
Specifies the maximum length for strings to be traced. The parameters for an RPC call are traced in the Data/Parameters column of the Trace Viewer's list. Parameters that are text or streams may be of any length,

and so must be truncated for display purposes. The default for Max String Length is 64, which should be suitable for the majority of users.

Inter-machine Sockets Dialog

The Inter-machine Sockets dialog is launched when the RPC Diagnostics - Sockets button in the Trace Viewer's tool bar is clicked. This dialog presents information on sockets and machine nodes.

Note: This dialog presents the same information that was formerly presented in the RPC Diagnostics application's MachineNode display pane.



Each line displayed in the Inter-machine Sockets dialog displays the following information:

- Name and IP address of each machine
- State of each machine (either remote or local)
- Dynamic display of the internal queue length
- Connection status (either green for a working connection, or red for a connection failure)

You may expand any line to view more detailed information about the inter-machine sockets by clicking the arrow icon to the right of the line. (Clicking the arrow icon a second time will collapse the selected line.) The expansion shows the individual sockets available for each remote machine connection.

Inter-machine Sockets Data for Remote Machines

The following data is displayed for each remote connection:

Discards	The total number of messages thrown off the queue.
Floods	The total number of full queue exceptions
Execs	The total number of RPCs executed by this machine node
RemVersion	The version of VTScada running on this machine
Session	The status of the RPC session (either Open or Closed)
Lost	The total number of times communications were irrecoverably lost
SocketsOK	The number of open sockets
Resends	The total number of message floods

Inter-machine Sockets Data for the Local Machine

Discards	The total number of messages thrown off the queue.
Floods	The total number of full queue exceptions
Execs	The total number of RPCs executed by this machine node
CtxErrs	Total number of RPC that failed due to a bad execution context being specified
ModErrs	Total number of RPC that failed due to a bad target module name being specified
TxRPC	Total number of RPCs transmitted
RxRPC	Total number of RPCs received
Pings	Total number of pings received
Acks	Total number of acknowledgments received
Disc	Total number of times this socket has been lost
Lost	Total number of times communications have irrevocably been lost
Coll	Total number of collisions encountered
Sync	The number of bad sync string errors received

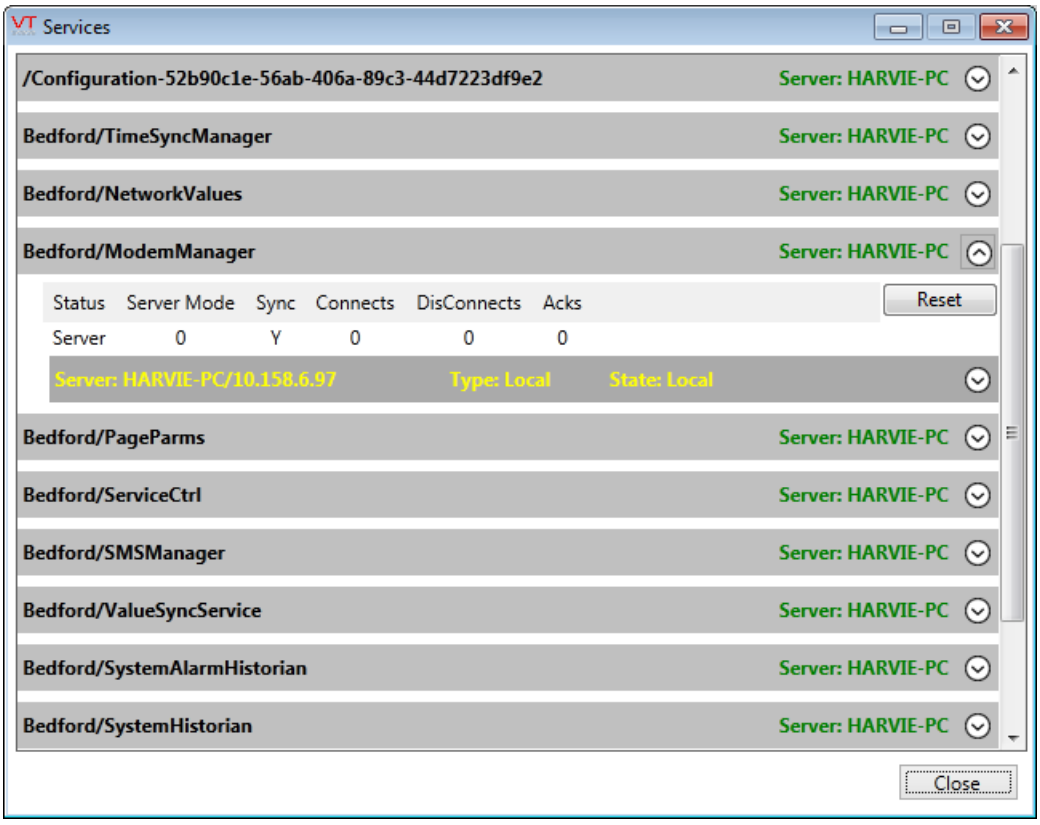
Hdr	The number of bad header errors received
Len	The number of bad length errors received
Pkt	The number of incomplete packet errors received

The Reset button to the far right of each socket statistics line can be clicked to reset the statistics.

Services Dialog

The Services dialog is launched when the RPC Diagnostics – Services button in the Trace Viewer's tool bar is clicked. This dialog provides information on services.

Note: This dialog presents the same information that was formerly presented in the RPC Diagnostics application's TagNode display pane.



Each line displayed in the Services dialog indicates the name of the application, the name of the service, and the name of the server for each service. In the example above, the last line displays "Completed Tutorial Example/Simulate_IO", indicating the Simulate_IO service in the

Completed Tutorial Example application. The server for this Simulate_IO service is displayed to the right (in this example, Anderson). Green is a quick visual indicator of those services where the current server is the local machine, while red indicates a networked machine.
You may expand any line to view more detailed information about the service by clicking the arrow icon to the right of the line. (Clicking the arrow icon a second time will collapse the selected line.)
The expansion shows the status of each machine participating in this service from the perspective of the local machine. In the above example, the ModemManager service in the Completed Tutorial Example application has two participating machines:
Dave (IP 192.168.1.46) is the local machine
Anderson (IP 192.168.1.33) is a remote machine

Information Displayed for a Local Machine

To the right of each participating machine identifier appears the type (either Remote or Local) and the state of the machine. If the machine is a local machine, the state can be one of:

Local	Indicates that this machine is the server
LocalIdle	Indicates that this machine is not the server

Information Displayed for a Remote Machine

If the machine is a remote machine, the state can be one of:

Machine Pending	Waiting for a socket connection to remote
AwaitSessionID	Got socket, waiting for RPC session to service
Connecting	Establishing connection with remote service
Syncing	Syncing with remote service
Connected	Good working connection to remote service

Any other reported values are transient.
Below the machine identifiers will be a series of columns that display connection information for each machine.

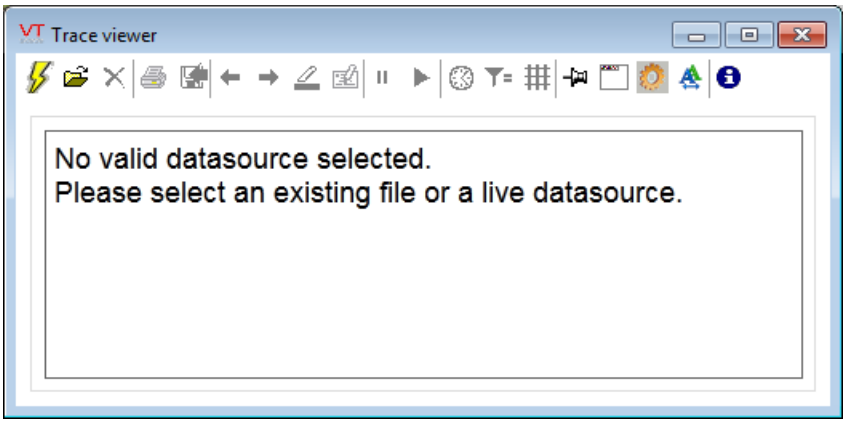
Information Displayed for a Client

If the identified machine is a client, the following information is displayed:

Status	The status of this machine, either Client or N/C (not connected)	
Client Mode	During service synchronization, RPC may set a non-zero mode here in order to filter the RPCs to be transmitted. The defined values are	
	0	RPC_ACCEPT_ALL (client is fully synchronized)
	64	RPC_ACCEPT_FILTER (mode cut off)
	128	RPC_SYNC_MODE (client is being synchronized)
	250	RPC_LINKCONTROL_ONLY (client requires synchronization)
Sync	Indicates the synchronization status of the client (Y indicates that this node is in sync with the server)	
Connects	A count of service connection attempts	
Disconnects	A count of service disconnects	
Acks	A count of service acknowledgments	

Using the Trace Viewer

The Trace Viewer is a script application included with VTScada. Before you can run the Trace Viewer, it must be added to the list of applications in the VAM.
If it does not appear in the VAM, you will need to add it.
On startup, the Trace Viewer will look similar to the following image. To begin using the functions, you must select either a live data source or a stored log file as prompted.



Related Tasks:

...Select a Live Data Source to View

...Select a Log File to View

...Clear the Current Trace

...Print the Trace Viewer's Data

...Export Data from the Trace Viewer

...Highlight Records

...Annotate Records

...Navigate to the Previous or Next Mark

...Pause and Run the Live Display

...Toggle the Timestamp Display

Related Information:

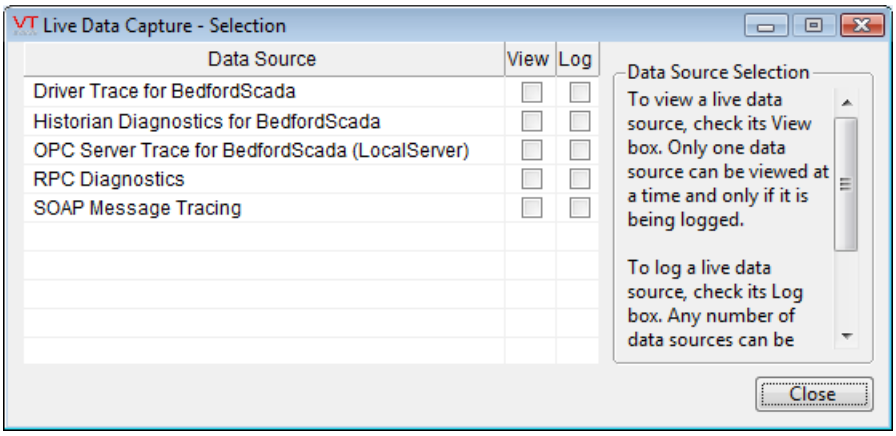
...Viewing vs. Logging a Data Source

...Trace Viewer Options and Controls

Select a Live Data Source to View

To select a live data source for viewing in the Trace Viewer:

1. Ensure the application you wish to examine is running.
2. Select the Attach to Live Capture button.  The Live Data Capture dialog will open and display the available data sources.



3. Select the name of the data source you wish to view.
   The data source will be highlighted to indicate that it is the one being viewed, and a checkmark icon will appear in the Log column to its right, indicating that this data source is running.
4. Select the Close button.
   The Trace Viewer dialog's list will be populated with data.

Related Tasks:

...Select a Log File to View

...Clear the Current Trace

...Print the Trace Viewer's Data

...Export Data from the Trace Viewer

...Highlight Records

...Annotate Records

...Navigate to the Previous or Next Mark

...Pause and Run the Live Display

...Toggle the Timestamp Display

Viewing vs. Logging a Data Source

There are two check boxes for each data source: View and Log. You can log data without viewing it, but cannot view without logging. Data must be collected in order to be displayed.

Since each data source is collecting different types of information, only one data source may be viewed at a time. You may simultaneously log as many as you wish.
It is possible to continue logging data after closing the Trace Viewer. This enables you to collect traces over a period of time when you suspect that an intermittent problem may be occurring. You should note however that trace files can quickly become large. Leaving the Trace Viewer running on a permanent basis is strongly discouraged.

Related Tasks:

...Select a Live Data Source to View

...Select a Log File to View

Select a Log File to View

To review trace data stored in a log file, follow these steps:

1. Select the Open File button  (or press Ctrl + O).
   The Open File dialog will open to the TraceFiles directory, and will display the available data files. (If the Open File dialog is not pointed to the correct directory, browse to the TraceFiles directory within the VTScada installation directory.)
2. Select the file you wish to view in the Trace Viewer.
   Driver trace files may have somewhat cryptic names, beginning with "Driver-DBTrace-". You may find that the date of the file gives the best indication to help choose which to open.
3. Select the Open button.
   The Trace Viewer dialog's list will be populated with data.

Related Tasks:

...Select a Live Data Source to View

...Clear the Current Trace

...Print the Trace Viewer's Data

...Export Data from the Trace Viewer

...Highlight Records

...Annotate Records

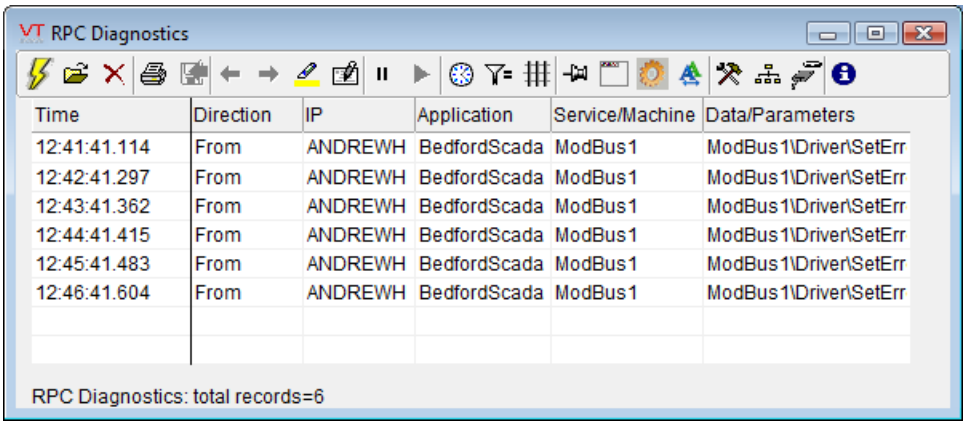
...Navigate to the Previous or Next Mark

...Pause and Run the Live Display

...Toggle the Timestamp Display

Trace Viewer Options and Controls

Data in the Trace Viewer display will look similar to the following image:



Note the row of control buttons across the top of the screen. The buttons displayed will depend on the data source selected, but most are common to all data sources. The common functions will be described first in the following sections.

Related Tasks:

...Clear the Current Trace

...Print the Trace Viewer's Data

...Export Data from the Trace Viewer

...Highlight Records

...Annotate Records

...Navigate to the Previous or Next Mark

...Pause and Run the Live Display

...Toggle the Timestamp Display

Information Displayed for a Server

If the machine is a server, the following information is displayed:

Status	The status of this machine, either Server or N/C (not connected)
Server Mode	During service synchronization, RPC may set a non-zero mode here in order to filter the RPCs to be transmitted. The defined values are: 0 RPC_ACCEPT_ALL (server not performing synchronization with any client) 64 RPC_ACCEPT_FILTER (Mode cut off) 128 RPC_SYNC_MODE (server is performing synchronization with a client) 250 RPC_LINKCONTROL_ONLY (server is starting synchronization with a client)
In Sync	Indicates the synchronization status of the server (Y indicates that this node is in sync with the server)
Srv/Sync	Indicates whether the server reports itself as in sync (Y indicates that the server is reporting itself in sync)
Alternate	Indicates whether a RecommendAlternate RPC call has been actioned (Y indicates that a RecommendAlternate RPC call has been actioned)
Session ID	RPC Manager maintains a session ID for each instance of an application running on a remote workstation (see "Session Ids " for further information).

Clear the Current Trace

To clear a current trace, select the Clear Current Trace button.  The trace you are viewing will be cleared and all logged data will be deleted from the trace file being viewed.

Note: Clearing a trace file is a permanent and irreversible action.

Related Tasks:

...Select a Live Data Source to View

...Select a Log File to View

...Print the Trace Viewer's Data

...Export Data from the Trace Viewer

...Highlight Records

...Annotate Records

...Navigate to the Previous or Next Mark

...Pause and Run the Live Display

...Toggle the Timestamp Display

Print the Trace Viewer's Data

The Trace Viewer enables you to print its data to any printer on the local PC's network. To print trace data follow these steps:

1. Select the Print button.  The Print dialog will open.
2. Set the printing parameters as you require (i.e. select the printer, page range, number of copies, etc.).

Note: The recommended print layout for the Trace Viewer is landscape.

3. Select the Print button.

Related Tasks:

...Select a Live Data Source to View

...Select a Log File to View

...Clear the Current Trace

...Export Data from the Trace Viewer

...Highlight Records

...Annotate Records

...Navigate to the Previous or Next Mark

...Pause and Run the Live Display

...Toggle the Timestamp Display

Export Data from the Trace Viewer

This topic is included for the benefit of anyone looking for this feature. An export option is not included since all data shown in the Trace Viewer is always being saved to a log file.

Related Tasks:

...Select a Live Data Source to View

...Select a Log File to View

...Clear the Current Trace

...Print the Trace Viewer's Data

...Highlight Records

...Annotate Records

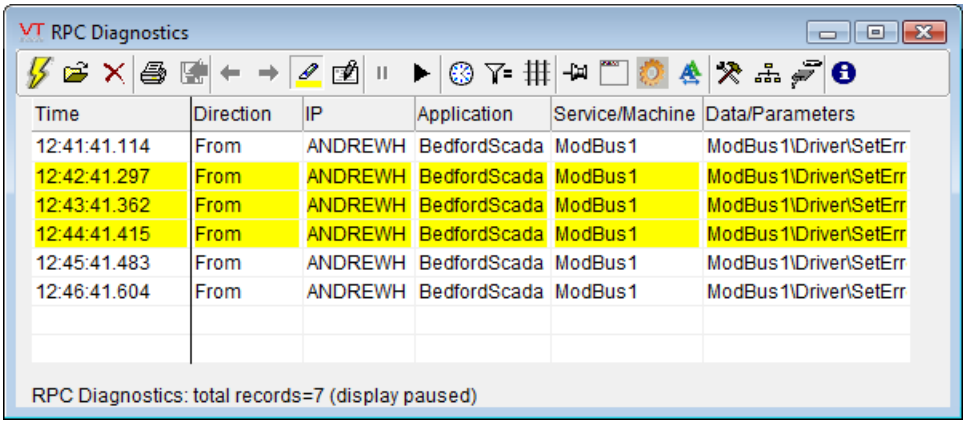
...Navigate to the Previous or Next Mark

...Pause and Run the Live Display

...Toggle the Timestamp Display

Highlight Records

You can highlight records of interest in the trace viewer. Highlighted records will appear similarly to the example shown. Note that performing this action causes the live display to pause. Data will still be logged, but you will need to resume the live display to see it.



To highlight records, follow these steps:

- Select the Highlight Records button.  The mouse cursor will appear as a highlight pen.
- Select the records you wish to highlight. The records will be highlighted in yellow.
- Select the Highlight Records button again to stop highlighting. The mouse cursor will revert to an arrow.

To Clear the Highlighting from a Record

Highlights can be cleared by following the same procedure you used to set them. The highlight pen works as a toggle: select a record once to highlight it, then select the record a second time to remove the high-light.

Note: The Trace Viewer enables you to navigate backward and forwards through the list to view highlighted items. Please refer to Navigating to a Mark in the Trace Viewer for instructions.

Related Tasks:

...Select a Live Data Source to View

...Select a Log File to View

...Clear the Current Trace

...Print the Trace Viewer's Data

...Export Data from the Trace Viewer

...Annotate Records

...Navigate to the Previous or Next Mark

...Pause and Run the Live Display

...Toggle the Timestamp Display

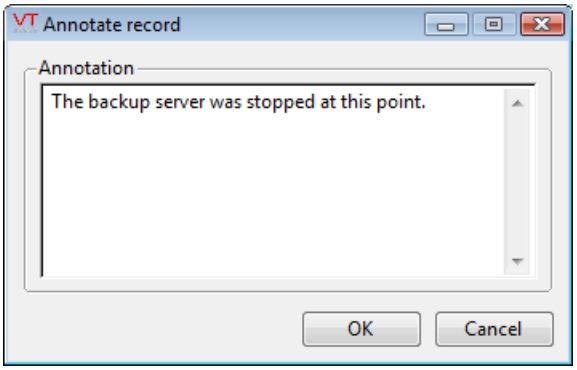
Annotate Records

The Trace Viewer enables you to add operator notes to records. The notes are stored in the database with the log data but, unlike operator

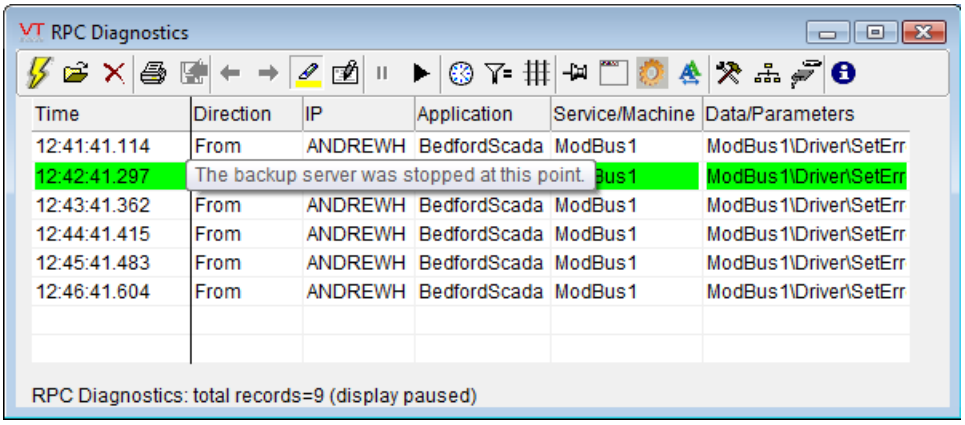
notes created in the Historical Data Viewer, these notes may be edited or deleted.

To add a note, follow these steps:

1. Select the Annotate Record button. 
2. Select the record you wish to annotate in the list. The Annotate Record dialog will open.
3. Enter the note in the field provided.



4. Click OK. The record with which the note is associated will be highlighted in green, and a window displaying the note's text will appear in a tool tip when the mouse pointer is rested on the record.



To edit or erase a Note:

Notes can be editing by following the same procedure you used to create them. Select the Annotate button, and then select a green record in the Trace Viewer. The Annotate Record dialog will re-open with the existing note displayed. Edit as you wish, then select OK.

Related Tasks:

...Select a Live Data Source to View

...Select a Log File to View

...Clear the Current Trace

...Print the Trace Viewer's Data

...Export Data from the Trace Viewer

...Highlight Records

...Navigate to the Previous or Next Mark

...Pause and Run the Live Display

...Toggle the Timestamp Display

Navigate to the Previous or Next Mark

The Trace Viewer application enables you to emphasize rows in the list with yellow highlighting. (see Highlighting Records in the Trace Viewer). When you have multiple items highlighted, you may navigate between them using the Go To Previous Mark  and Go To Next Mark  buttons.

To view a previous mark in the Trace Viewer's list, select the Go To Previous Mark button. The list will display the last item that was highlighted in the list. Continuing to select the Go To Previous Mark button will take you through the highlighted items in the list until you have viewed them all. The Go To Previous Mark button will be disabled when you have viewed all previously highlighted items.

The Go To Next button works in a similar way.

Related Tasks:

...Select a Live Data Source to View

...Select a Log File to View

...Clear the Current Trace

...Print the Trace Viewer's Data

...Export Data from the Trace Viewer

...Highlight Records

...Annotate Records

...Pause and Run the Live Display

...Toggle the Timestamp Display

Pause and Run the Live Display

You can pause and resume the display of live data in the Trace Viewer using the Pause Live Display  and Resume Live Display  buttons. Pausing the live data display enables you to take the time to closely examine records in the list, especially in large applications where tracing moves quickly.

Note: Pausing the live display does not stop tracing and logging from continuing.

Related Tasks:

...Select a Live Data Source to View

...Select a Log File to View

...Clear the Current Trace

...Print the Trace Viewer's Data

...Export Data from the Trace Viewer

...Highlight Records

...Annotate Records

...Navigate to the Previous or Next Mark

...Toggle the Timestamp Display

Toggle the Timestamp Display

The first column of the Trace Viewer's list can be toggled to display one of three formats:

- Time: The Time column displays the time to the nearest millisecond for each item in the list.

- Date/Time: The Date/Time column displays the date in the format MM/DD/YYYY (e.g. 11/29/2005), and the time to the nearest millisecond for each item in the list.
- Record: The Record column displays a record number for each item in the list. Record numbers assist users in easier reading of the list by sequentially numbering items.

To switch between these options, select the Toggle Timestamp Display button.  The column's label and data will change to match.

Related Tasks:

...Select a Live Data Source to View

...Select a Log File to View

...Clear the Current Trace

...Print the Trace Viewer's Data

...Export Data from the Trace Viewer

...Highlight Records

...Annotate Records

...Navigate to the Previous or Next Mark

...Pause and Run the Live Display

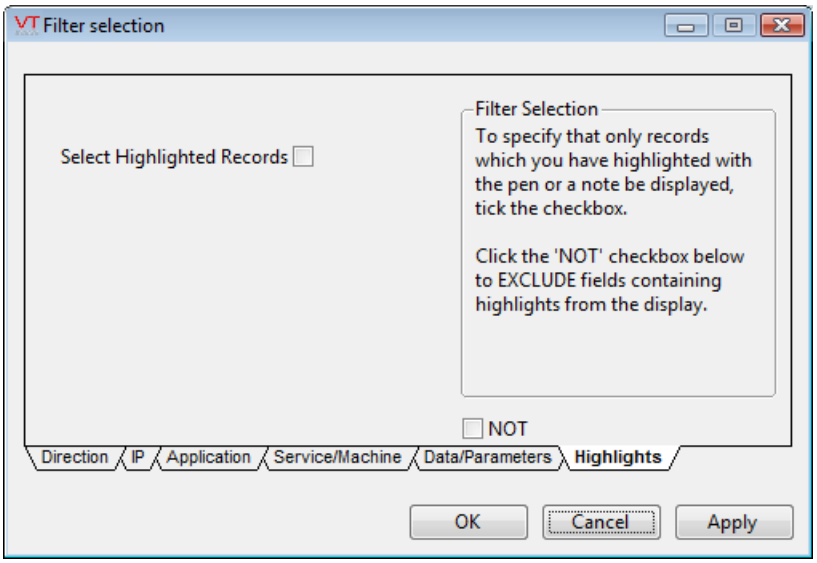
Filter the Trace Viewer's List

The Trace Viewer enables you to filter the data being displayed in the list so that you may analyze only the information that you are interested in. Note that the filter applies only to the display: all data continues to be logged.

Which columns can be filtered and what criteria are available for filtering both depend on which live data source you are viewing. As a minimum, the ability to filter for only the highlighted records is always available. Other filters may allow multiple selections from a list of options, or free text input that will be searched for in the specified field.

In general, to filter the Trace Viewer's display list:

1. Select the Define Filter button. The Filter Selection dialog will open, showing the options appropriate to the type of trace you are viewing. The example shown here is for the Driver trace.



2. Set the filtering options you require (see Filter Selection Dialog Options for detailed instructions).
3. Select the Apply button. The list will be filtered according to the criteria you specified.
4. Continue to adjust the filtering parameters as you require.
5. Select the OK button to close the Filter Selection dialog.

To remove filters from the Trace Viewer's list:

Filters are removed the same way they are set. In the Filter Selection dialog, for each tab in which you have set a filter, open the tab and clear the filters from the list by selecting them and using the Delete key on your keypad.

Related Information:

...Filtering Options

Related Tasks:

...Select a Live Data Source to View

...Select a Log File to View

...Clear the Current Trace

...Print the Trace Viewer's Data

...Export Data from the Trace Viewer

...Highlight Records

...Annotate Records

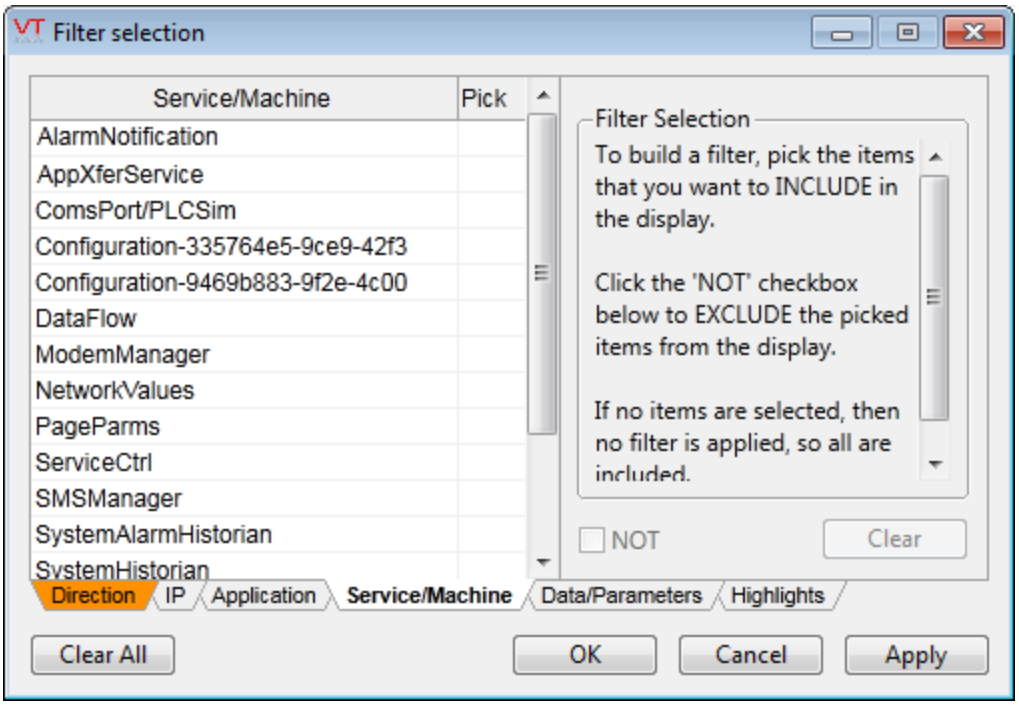
...Navigate to the Previous or Next Mark

...Pause and Run the Live Display

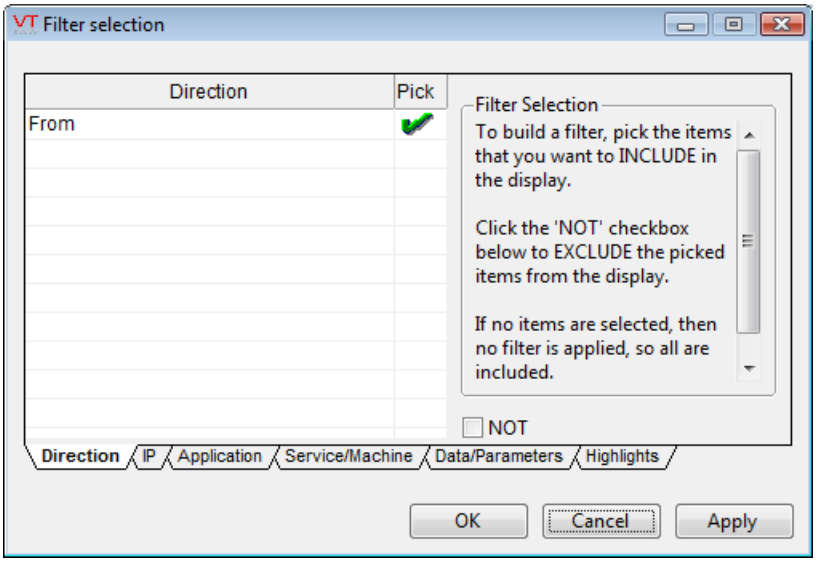
...Toggle the Timestamp Display

Filtering Options

 When a filter is active, this button will have an orange highlight: 
A filter can be applied to almost every column in the Trace Viewer display. Filters follow one of two formats. They will either provide a list of possible values that you can filter for, or a text box where you can enter words that the column must contain.
In every case, there will be a NOT option that enables you to reverse the effect of each filter.
The Filter Selection dialog provides a number of options for filtering the Trace Viewer’s list. The elements of the Filter Selection dialog are described here to help assist you in filtering data. Tabs that contain active filters are highlighted.



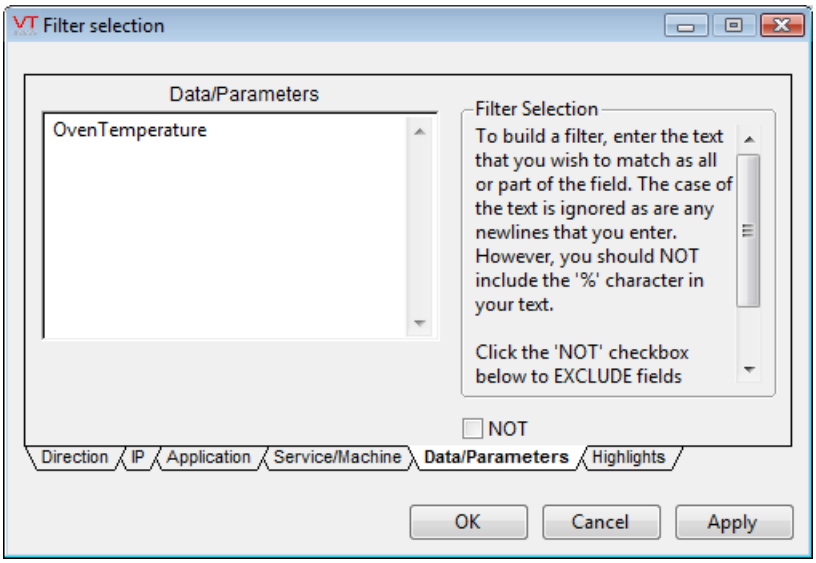
Example of a Pick Filter (Direction)



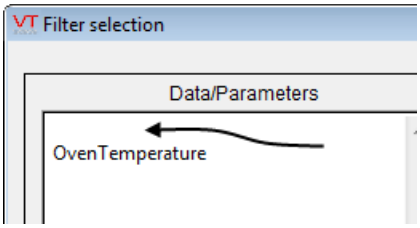
For every instance of a pick filter, a list of the available values for each column will be displayed in the dialog. Select each value you want to filter for (or exclude, if using the NOT option) and then select the Apply button. You select an option by clicking in the Pick column.

The filter is cleared by removing all of the checkmarks from the Pick column.

Example of a Text Filter (RPC Data/Parameters)



For every instance of a text filter, enter the words you wish to filter for (or exclude, if using the NOT option) in the space provided. Partial words are acceptable but wildcards are not. Spaces count as characters to be filtered for. New line characters are ignored unless the new line character is the first character in the filter. Note the extra line in the following image. The leading new line in this example will cause the filter to fail to find any matches. Leaving a new line character behind when clearing a filter is an easy mistake to make.



Select Columns for Display in the Trace Viewer's List

You can choose to enable or disable the display of all columns except the timestamp. Some columns that you may wish to view are not displayed by default. Also, you might decide that some of the columns are not relevant to the information you are interested in.
To modify which columns are displayed:

1. Click on the Select Displayed Columns button.  The Select Displayed Columns dialog will open.

2. Select the Show column to hide or show columns. Any column names that display a checkmark in the Show column on their right will be displayed, while those column names that do not display a checkmark in the Show column to their right will be hidden.
3. Select the OK button. The Trace Viewer's list will display only those columns that were selected for viewing.

Related Information:

...Trace Viewer Visibility and Display Options

Related Tasks:

...Select a Live Data Source to View

...Select a Log File to View

...Clear the Current Trace

...Print the Trace Viewer's Data

...Export Data from the Trace Viewer

...Highlight Records

...Annotate Records

...Navigate to the Previous or Next Mark

...Pause and Run the Live Display

...Toggle the Timestamp Display

Trace Viewer Visibility and Display Options

Keeping the Trace Viewer On Top of Other Windows

To keep the Trace Viewer dialog on top of all other windows, select the On Top Of Other Windows button. 

Hiding and Revealing the Trace Viewer's List

You can hide and reveal the Trace Viewer's data display by selecting the Toggle Toolbar View button in the Trace Viewer's tool bar. 

Modifying Trace Viewer Settings

To modify the settings for the Trace Viewer, click the Viewer Settings button. The Trace Viewer Settings dialog will open and allow you to modify the following settings:

Refresh Rate – Server: You can modify the rate at which the Trace Viewer updates its display from the server using this spin box. The default refresh rate is 1 second.

Refresh Rate – VIC: You can modify the rate at which the Trace Viewer updates its display over a VIC connection using this spin box. The default refresh rate is 5 seconds.

Translate IPs to Names: You can select this check box to have the Trace Viewer translate IP addresses into NetBIOS names.

Modifying the Font Used in the Trace Viewer's List

If you find that the data being displayed in the Trace Viewer's list is difficult to read, you may modify the font used to display list items.

To modify the font used to display items in the Trace Viewer's list:

1. Click the Select Font button.  The Font dialog will open.
2. Select the typeface you wish to use from the Font list.
3. Select the style you wish to use from the Style list.
4. Select the size you wish to use from the Size list.
5. Click the Open button. The Trace Viewer's list will be adjusted to use the selected font.

Displaying the Trace Viewer's version information

You can find which version of the trace viewer you have installed by selecting the About button from the menu. 

Trace VTScada Actions Application

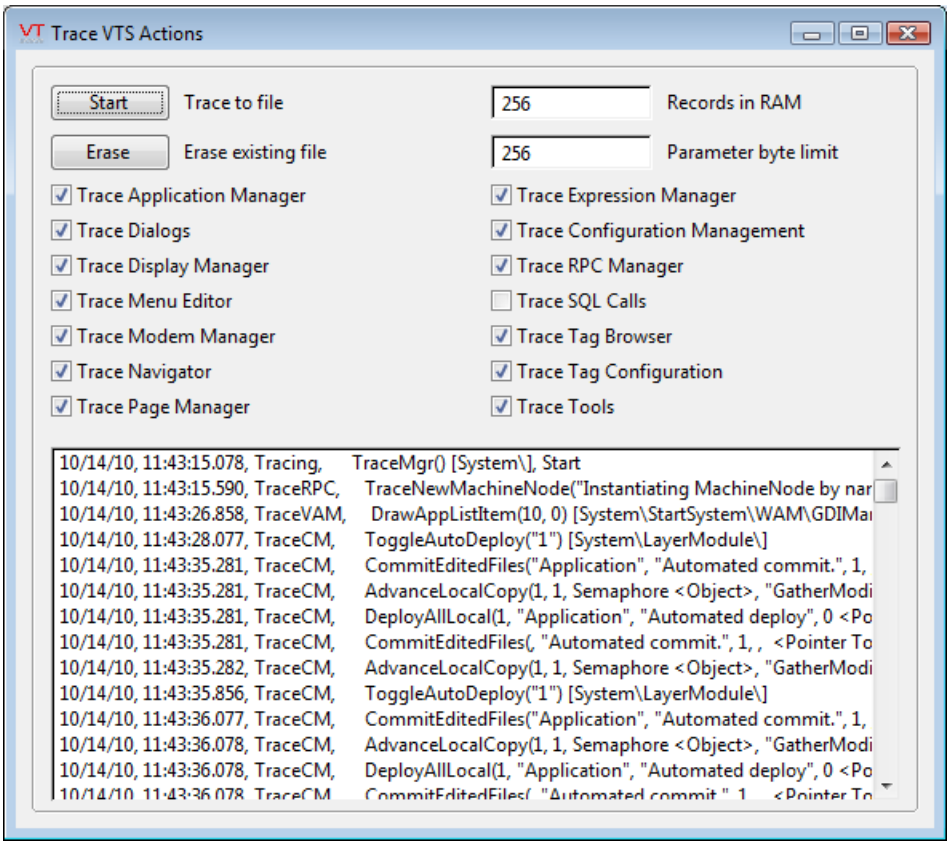
Note: The Trace VTScada Actions application should not be confused with the Trace Viewer application that permits you to view trace

information for your applications in real-time.
This application is included with every copy of VTScada, but you may need to add it to the VAM.

The Trace VTScada Actions script application enables you to select different VTScada services (such as the RPC Manager and Modem Manager), and actions (the Navigator or SQL calls), and monitor the selected items by saving pertinent data about them (such as the date and time they executed) to disk.
Service names that contain a child-tag delimiter will be shown with a forward-slash.

Note: The Trace VTScada Actions application may still be used to write data about selected actions to a text file named, "VTSTrace.txt". Additionally, VTScada traces all actions to disk, storing the data in a separate text file named, "VTSTraceAll.txt". Both the VTSTraceAll.txt and VTSTrace.txt files are automatically written to the VTScada installation directory when you exit your application.

An example of the Trace VTScada Actions dialog:



The Trace VTScada Actions utility consists of the following elements:

Start/Stop

The Start/Stop button enables you to start and stop the tracing of the selected actions to a special text file named, "VTSTrace.txt" (located within your VTScada installation directory (e.g. C:\VTScada\VTSTrace.txt). When the button is toggled on (depressed and labeled "Stop"), VTScada is tracing the specified actions to file. When the button is toggled off (labeled "Start"), VTScada has stopped tracing the specified actions to file. As mentioned in the note above, by default, VTScada now traces all actions to a "VTSTraceAll.txt" file, located in your VTScada installation directory.

Erase

The Erase button deletes the tracing file (VTSTrace.txt) from disk when actions have been traced to file using the Start button. The Erase button is disabled while the Start button is selected.

Records in RAM

The Records in RAM field enables you to specify the number of trace actions you wish to be saved to RAM prior to being written to the VTSTrace.txt file. The default for Records in RAM is 256.

Parameter Byte Limit

The Parameter Byte Limit field enables you to specify the maximum size of parameters. The default for Parameter Byte Limit is 256.

Trace Application Manager

The Trace Application Manager check box can be selected to indicate that you wish all activities pertaining to the VAM to be traced.

Trace Dialogs

The Trace Dialogs check box can be selected to indicate that you wish all activities pertaining to dialogs to be traced. This setting is useful if you wish to trace 4BtnDialog calls.

Trace Display Manager

The Trace Display Manager check box can be selected to indicate that you wish to record all activities related to the Display Manager.

Trace Menu Editor

The Trace Menu Editor check box can be selected to indicate that you wish to trace all activities related to the Menu Editor.

Trace Modem Manager

The Trace Modem Manager check box can be selected to indicate that you wish to trace all activates related to the Modem Manager.

Trace Navigator

The Trace Navigator check box can be selected to indicate that you wish to trace all activities related to the shortcut menus in your application.

Trace Page Manager

The Trace Page Manager check box can be selected to indicate that you wish to trace all activities related to the Page Manager.

Trace Expression Manager

The Trace Expression Manager check box can be selected to indicate that you wish to trace all calculations related to the Expression Manager.

Trace Remote Configuration

The Trace Remote Configuration check box can be selected to indicate that you wish to trace all activities related to remote configuration.

Trace RPC Manager

The Trace RPC Manager check box can be selected to indicate that you wish to trace all activities related to the RPC manager and remote procedure calls.

Trace SQL Calls

The Trace SQL Calls check box can be selected to indicate that you wish to trace all SQL calls made in your application.

Trace Startup Synchronization

The Trace Startup Synchronization check box can be selected to indicate that you wish to trace all activities related to start-tup synchronization.

Trace Tag Browser

The Trace Tag Browser check box can be selected to indicate that you wish to trace all activities related to the Tag Browser.

Trace Tag Configuration

The Trace Tag Configuration check box can be selected to indicate that you wish to trace all activities related to tag con-figuration.

Trace Tools

Obsolete as of VTScada release 11.

Note: Each of the check boxes described above corresponds to a con-figuration variable in the system-wide Setup.ini file (stored in the VTScada installation directory). Information on these variables can be found in "Configuring Setup.ini".

Trace List

The list displayed on the Trace VTScada Actions dialog displays the actions that have been traced according to the check boxes you've selected. For each action traced in the list, the following data is displayed:

Date: The date for each action being traced is displayed in the first column of the trace list. The date is displayed using the format, "MM/DD/YY" (e.g. 10/24/05).

Time: The time at which each action was traced is displayed in the second column of the trace list. The time is displayed in milliseconds, using the format, "HH:MM:SS.MS" (e.g. 16:07:23.129).

Trace Action: The category of the action being traced is listed in the third column, according to the check boxes you have selected (e.g. TraceVAM or Trace RPC).

Data: The details about each action being traced are identified in the fourth column.

Note: In the event that some items in the list are difficult to read, you can expand the Trace VTScada Actions dialog either using the Windows maximize button in its title bar, or by dragging its borders, or you can rest your mouse pointer over each entry in the list to view its details.

Related Information:

...Trace Viewer Application

Historian - API and Queries

The Historian Manager is responsible for data logging, the Historian tags for your application, and storage & retrieval of logged tag values. Versions of VTS prior to release 10 used a Log Manager Service, which is now obsolete. This chapter describes the Historian Manager, data storage options for the Historian and how to configure VTScada to read legacy data that was created using the Log Manager service.

Related Information:

...Recording Data

...Historian Manager API

...VTScada SQLInterface Module

Recording Data

The default storage location for logged data is a file database system that was developed by Trihedral Engineering Ltd. Information is stored within the Data folder of your application. An API that provides read and write functions has been provided for use in your custom code.
In addition to, or instead of, the VTScada file database system, you can store data using any of the following database formats.

Note: There is no advantage to be gained in speed or reliability by using a third-party database. If your goal is to provide a way report on VTS data using a database format and tools that are familiar to you, then you are strongly advised to add SQL Logger tags to your application rather than change the fundamental storage system.

Supported databases:

- Oracle 10g or later
- SQLServer 2000 or later

- MySQL using the MySQL ODBC Connector 5.1.6 or later
- SQLite using the SQLite ODBC driver version 0.86 or later

The choice of database system to use is controlled by the application property, StorageType.
If using a StorageType other than the default, you must also tell VTScada where to find that database, using the StorageLocation property.

Related Information:

...Specify the Storage Type for Historian Data

...Specify the Location for Historian Data

Specify the Storage Type for Historian Data

If you are using the default file database system for your data, no configuration needs to be done. If you would like to save data to one of the four supported database system instead, you must set the StorageType property for each Historian tag that will use that system.
The actual name of the property used is a combination of the Historian tag name and the keyword "StorageType". Thus, to set a storage type for the SystemHistorian, you would set a value for "SystemHistorianStorageType". For a tag named HistorianA, "HistorianAStorageType".
This property should not be modified if using the VTScada data store. Otherwise, "ODBC" is to be used for all other database storage formats since the Historian relies on the ODBC for communication with each. The specifics for connecting to a particular database are described in the next section, Specify the Location for Historian Data.

Related Information:

...Specify the Location for Historian Data

Specify the Location for Historian Data

You may set a specific location for the data and log files for your application using the application property, StorageLocation. Although the

names of the properties are given here, these values should be set using the Historian tag's configuration panel.
The actual name of the property used is a combination of the Historian tag name and the keyword " StorageLocation ". Thus, to set a storage type for the SystemHistorian, you would set a value for "SystemHistorianStorageLocation ". For a tag named HistorianA, "HistorianAStorageLocation ".
The default value is "History". If setting an alternate storage location for a file database, you should provide the full path to the folder that you want to use. The most common reason for this is if you want to save data to a disk other than the one on which VTScada is running. This could also be done by mounting an alternate disk to the Data\History path via MS Windows™. The constant disk usage associated with logging data from a large or medium sized application may cause a disk to wear out faster than it otherwise would. By keeping the data store separate from the disk that is running VTScada, and maintaining redundant storage on another server, you can lessen the impact of disk failure on your operations.

Note: In tests, directing the Historian to save data to network share locations proved to be slow. Use caution if you intend to re-direct the location of the VTScada database to a .shared location on your network

If setting an alternate storage location for an ODBC database, you can provide either a Data Source Name (DSN) or a connection string.
The advantage of a DSN is that it is relatively easy to configure using the Microsoft ODBC Administrator™ dialog. The disadvantage is that you or your system administrator must create that DSN on each server that the application runs on.
A connection string is somewhat longer to create, but once written it may simply be copied to each server. Connection strings may be easier to maintain over the life of the application.
Support is also provided for FileDSNs.
Example using a DSN:

```
SystemHistorianStorageLocation = DSN=MyDSN_Name
```

Example using a connection string:

```
SystemHistorianStorageLocation = Driver=SQL Server; Server-
r=ServerName;Database=DBName;Uid=user;Pwd=password
```

Related Information:

...Specify the Storage Type for Historian Data

...Historian Tags are described in the VTScada Developer's Guide

---

Historian Manager API

The Historian Manager is a service that runs in the VTScada layer. It contains two public functions that can be accessed by your VTScada application. Note that script applications, which are not based on the VTScada layer, will not have access to these items.

Note: The GetLog function has been marked as deprecated. All legacy code using that function should be updated to use the Historian functions instead.

Data logging is done through the use of the WriteHistory function. Later retrieval of that data is done using the GetTagHistory function.

Related Information:

...Trending and Plotting Functions and Statements

...Data Logged or Trended Variables in Tag Modules

Related Functions:

... GetTagHistory

... WriteHistory

Trending and Plotting Functions and Statements

There are two statements within VTScada that perform plot operations: The first is the Plot statement, which plots an array of values against its index. This is the most common plot type and can be used to create a line plot, or a filled or bar plot (these types of plots are useful for filling

rectangular tanks with the trend of the recent tank levels as an alternative to just a bar of the current tank level).
The second plot statement is the PlotXY statement, which plots the values in one array against the values in a second array. This can be useful for plotting one plant parameter against another, such as plotting production rate against conveyer speed. One problem arises when plotting such values; since the values of both parameters normally both increase and decrease, the plot will appear as somewhat of a scatter if the X values are not consistently increasing. To solve this problem, the Sort statement can be used to re-order the arrays so that the Y values correspond to increasing X values.
The PlotXY statement has another possible use; it can be used as an alternative line drawing statement, with each of the array elements specifying a line segment endpoint.
VTScada enables you to configure various line styles and fill patterns for bar plots. The plots may also be arranged to plot from left to right, right to left, bottom to top, or top to bottom. There are a total of 8 possible orientations for line plots, and 16 for bar plots. Optionally, plots may be configured to display in a digital or discrete format; instead of drawing a straight line between two points, a step or square-looking plot is drawn that shows discrete changes in level, rather than continuous changes.
Another option enables a given bit number in the array to be plotted – this enables arrays of values that contain status bits packed into short or long values to be plotted directly without having to unpack them. A final option enables groups of consecutive array elements to be averaged and plotted as a single value on the screen; this enables a large amount of data to be plotted on a screen of limited resolution without producing a high degree of apparent scatter in the data and while improving the plot speed.

Data Logged or Trended Variables in Tag Modules

Any data logged or trended variables in a tag module must be declared with a class in the range of 1 to 6, using syntax similar to the following:

```
value (1);
```

The class specifies the type of data to log, as shown in the following table:

Class:	Data Type:
1	Bit
2	Unsigned Byte
3	16 Bit Signed Integer
4	32 Bit Signed Integer
5	Double Precision Floating Point
6	Text or Binary Data

Variables must be declared with an appropriate value or else they will not be logged.

The convention for declaring variables is alphabetically by variable class type. For example, if you were to declare Z(6), Y(2), X(5), A(5) then, internally this will map to Y(2),A(5), X(5), Z(6).

After data has been logged by a tag, if you then add, delete or change the type of the variables, the old logged data will become inaccessible (although, it is not deleted). Changing type of logged variables or adding/deleting is akin to creating a new tag for the purposes of the Historian database.

If it is permissible for the data to be trended but not logged, the tag must be added to the "Trenders" tag group. Such tags will show a limited amount of data when viewed via the HDV.

If you have advanced logging requirements and want to integrate logging behavior into your tag, you must do the following:

- Your tag must have a parameter or variable named "HistorianName". This should be set to the name of the HistorianTag to be used by your tag. It is better to use a parameter than a variable because parameters provide flexibility in configuration.
- Note that any tag with a HistorianName parameter or variable will be automatically added to the Loggers and Trenders tag groups.
- You must declare logged variables as stated above.

- You must call \HistorianManager\WriteHistory() whenever you want to log data.

Related Information:

...Logging Tag Data

VTScada SQLInterface Module

This service provides an SQL interface to VTScada historical data, current tag values, alarm data, or other custom tables on an application-by-application basis.
The interface can be accessed externally using SOAP or ODBC calls, providing support for a subset of SQL[1]. Within VTScada, you might use the SQLInterface for the convenience that the SQLQuery function provides in some instances, relative to making multiple calls to GetTagHistory.

Note: The SQLQuery function is essentially a wrapper for GetTagHistory. No SQL query parameters are supported other than those that can be performed by one or more calls to GetTagHistory and other VTScada functions. SQL functions that modify data or database structures are not supported.

The interface provides the following two functions:

- SQLQuery
  Executes an SQL query on data in a VTS application by turning that query into one or more calls to GetTagHistory.
- RegisterCustomTable
  Before SQLQuery can be work, RegisterCustomTable must be used to record what information is to be available for a defined table name and how values are to be retrieved. All VTScada tags, notes, and alarm information has already been registered for you. Use this function only for custom data inquiry needs.

---

[1]Structured Query Language

SQL queries specify table names where data is to be found. This does not match the system used by VTScada for data storage or retrieval, even if you have configured your Historian to use a third-party SQL database. There is no History table, nor is there an Alarms table, or any History_TPP[1] tables. But, the SQLQuery function is able to retrieve tag data as if those tables existed because RegisterCustomTable was used to link table names to the instructions for finding relevant data.

Related Information:

...SQLQuery - Function for retrieving logged data using structured query language

...RegisterCustomTable - Function for registering a custom table from which SQLQuery will retrieve information.

...SQL Queries of VTScada Data: The ODBC Server - VTScada Developer's Guide - Configuration and examples.

---

[1]Time Per Point. The time span used when querying aggregated data (average, minimum, maximum, etc.) from tag history.

Programming Other Modes of Communication

I/O device drivers are only one of the options available to you for linking your VTS application to external sources of data. You may also use COM (Component Object Model), DDE (Direct Data Exchange), TCP/IP, ODBC (Open Data Base Connectivity) or DLL (Dynamic Link Libraries).
Tools and techniques for using each of these technologies can be found in the topics within this chapter.

Related Information:

...Communicating Directly With Hardware

...Using COM in VTS

...Using DDE

...TCP/IP Networking

...Using ODBC

...Using DLLs

...See also: Communication Drivers - how to write a custom driver.

Communicating Directly With Hardware

VTScada, through the VTSIO driver, has the ability to interface directly with the memory addresses and IO ports of your computer's hardware. This is especially useful when you need to interact with legacy hardware for which there is no driver and software API.

Note: In order to write code that interfaces directly with memory addresses and I/O ports, you must have a detailed knowledge of the hardware in question. You should obtain the product specification sheets before attempting to configure the VTSIO driver.

Having an instance of the VTSIO driver configured to interface with a particular piece of hardware enables the following functions to be used to read and write data, to and from the hardware: MemIn, MemOut, CopyIn, CopyOut, In, InWord, Out, and OutWord. Guidance for using the can be found in the function reference, elsewhere in this Guide.

Related Information:

...Configuring a VTSIO Driver as the Interface to PC Hardware

...Configuring a single instance of the VTSIO driver:

Configuring a VTSIO Driver as the Interface to PC Hardware

In order to configure an instance of the VTSIO Driver, you must know the resources used by the hardware. This is generally a series of IO Port numbers, memory addresses, or both.
A single instance of the VTSIO Driver can handle only a single contiguous range of IO Port numbers or memory addresses. For example, a piece of hardware might use IO ports 250 through 257 and memory addresses D0000 through D3FFF. For this application, two instances of the VTSIO driver would be required – one for the IO port range and one for the memory address range.

Configuring a single instance of the VTSIO driver:

Given a contiguous IO port range or memory address range, the following steps will configure an instance of the VTSIO driver. Here, the VTScada installation directory is assumed to be C:\VTS. If a different name has been used for the VTScada installation directory, substitute as required.

1. Create a folder for the driver files (e.g. c:\DriverInstance1).
   The following steps will refer to this as "the driver folder". This folder may be deleted at the completion of the steps, or you may keep it in case the driver needs to be re-installed.
2. Copy the appropriate .inf file from the VTScada installation's Template directory to the driver folder.

- IO.INF for an IO port range
- MEMORY.INF for a memory address range

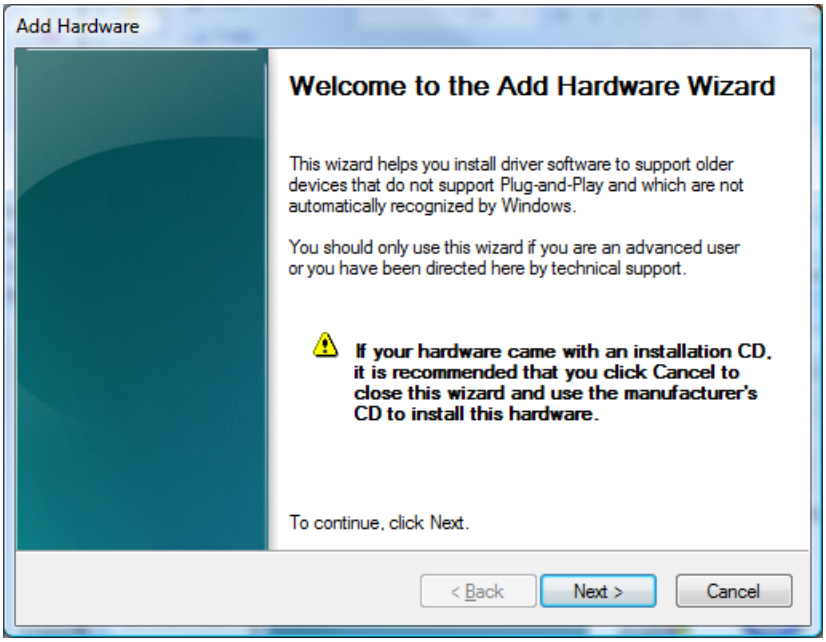
3. Copy the subdirectories i386 and amd64 from C:\VTScada\DRIVER\VTSIO\ to the driver folder.
4. Edit the .inf file copy that you copied to the driver folder, specifying the IO port or memory address range.
   - For an IO port range, change the number range specified on the line that starts with IOConfig=.
   - For a memory address range, change the number range specified on the line that starts with MemConfig=.

Note that these numbers are hexadecimal. Save the edited file.

The following instructions are for Windows Vista; the process is very similar for Windows XP.

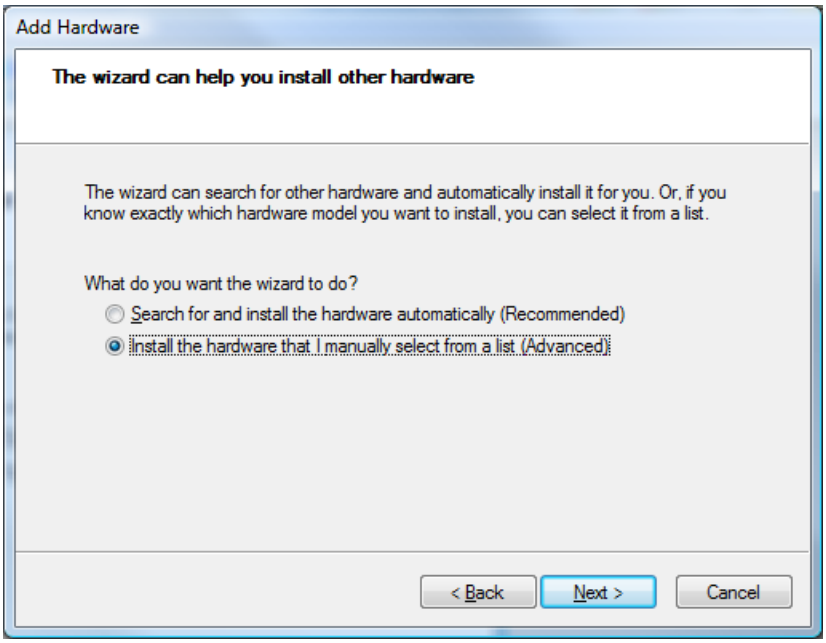
5. In the Control Panel (in Classic View), double-click on Add Hardware.

The Add Hardware Wizard should appear.

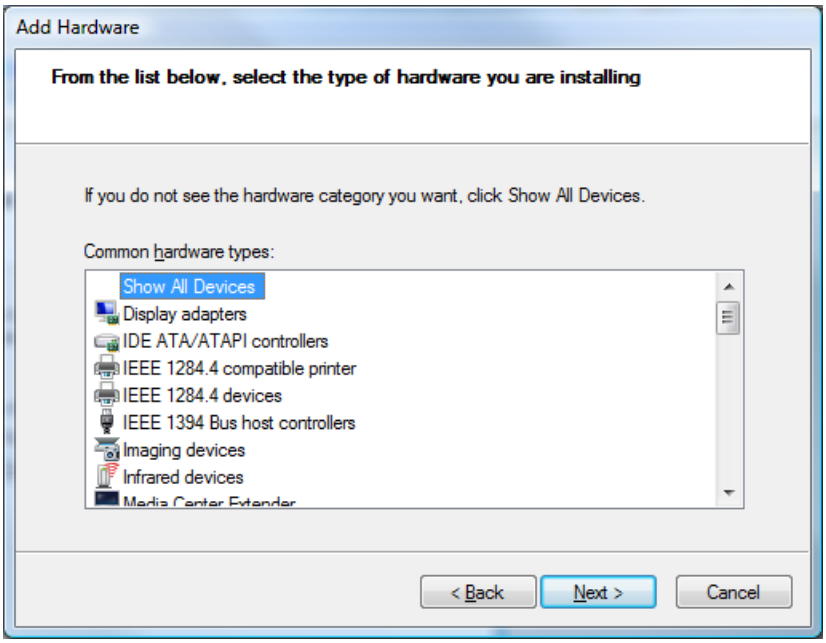


6. Click the Next button.

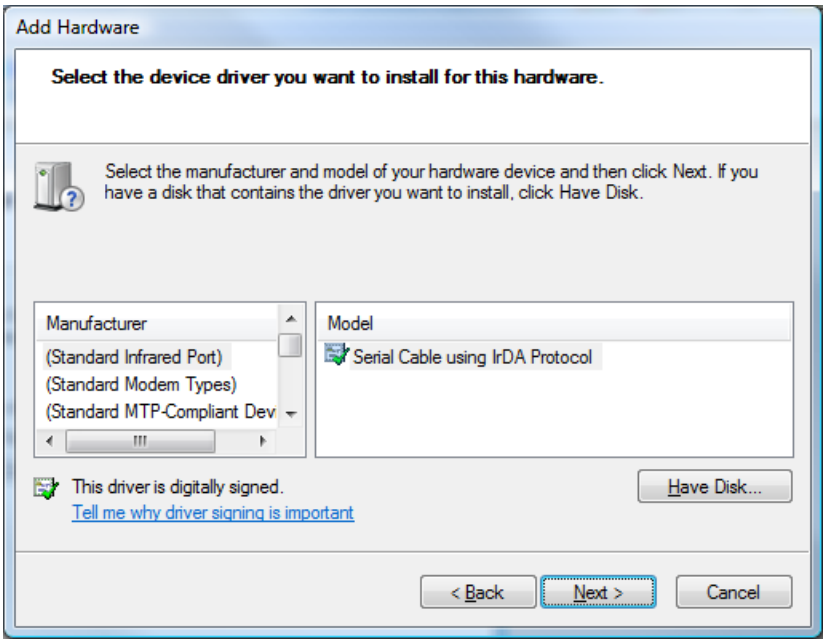
The dialog should look like the following:



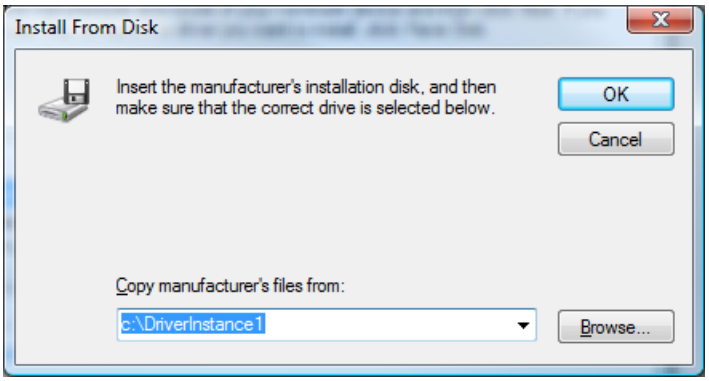
7. Select "Install the hardware that I manually select from a list (Advanced)" and click Next.



8. Leave "Show All Devices" selected and click Next.



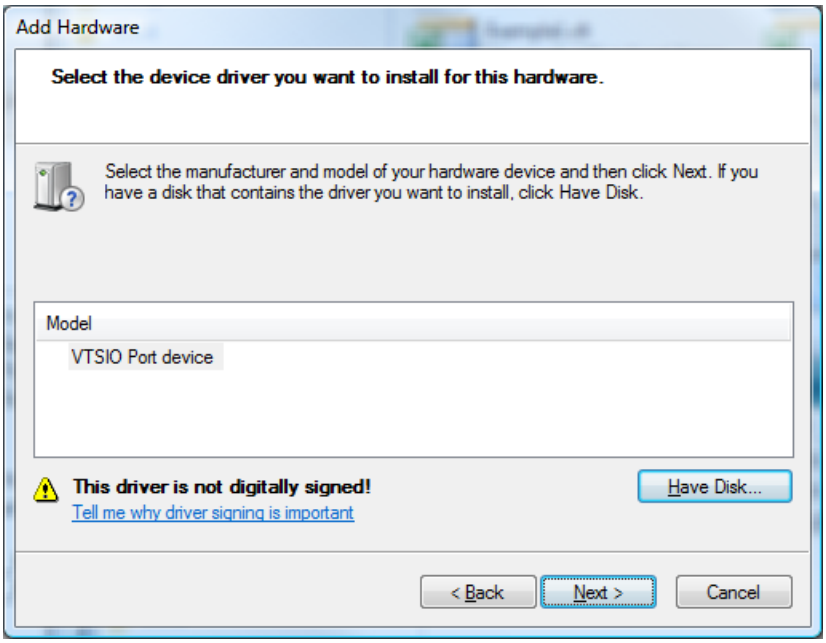
9. Without changing anything in the list boxes, click the "Have Disk..." button.



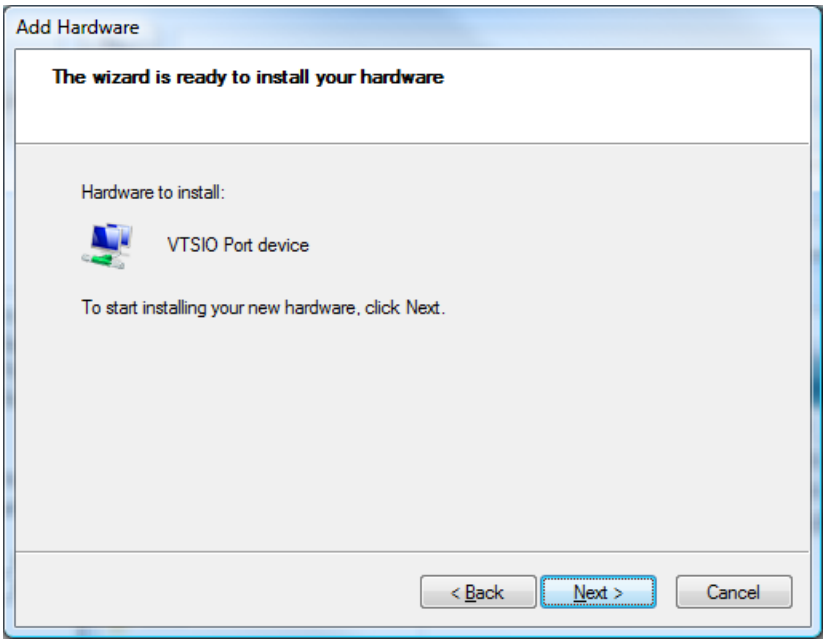
10. In the box labeled "Copy manufacturer's files from", enter the path to the driver folder, or browse to that directory.

    If you used the suggested name, then that directory will be "c:\Driver-Instance1", as in the above screenshot. Click OK.

    The dialog now should look like the following if you are creating a VTSIO driver for an IO port range, using IO.inf (this dialog, and dialogs that follow, will say VTSIO Memory device if you are using memory.inf).



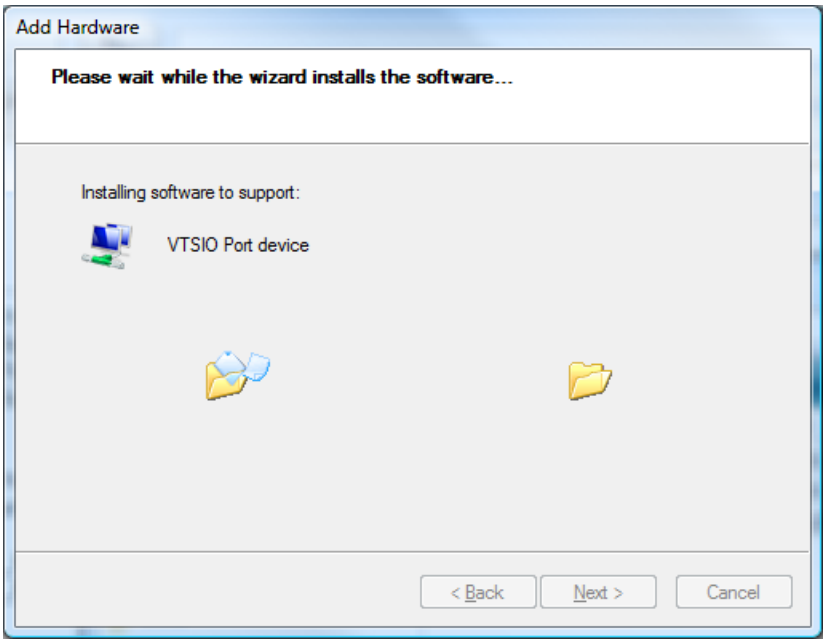
11. Click the Next button.



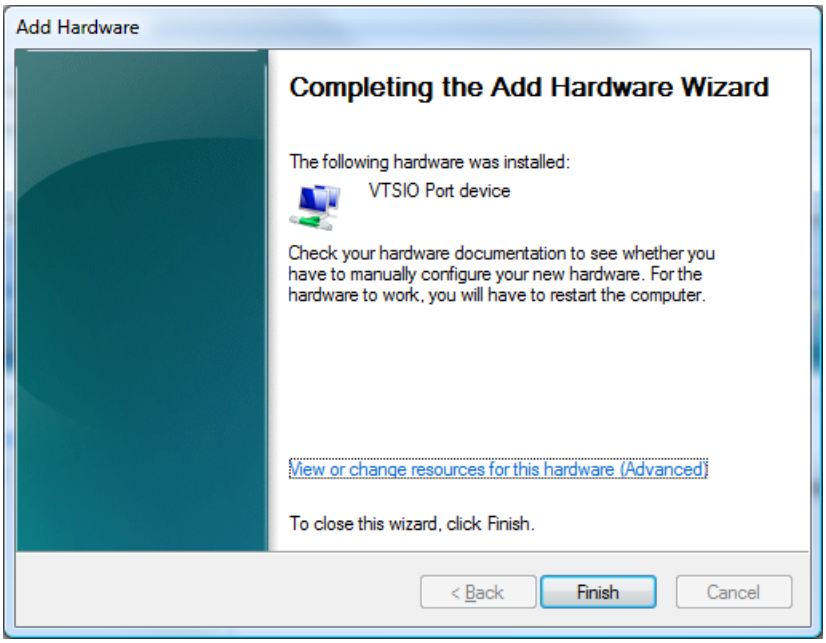
12. Click the Next button.

    If you get a warning about the driver not being digitally signed, select "Install this driver software anyway".

    You should see a driver installation status dialog like the following while the driver is being installed.



When the driver has finished installing, the dialog should look like the following.



13. Click the Finish button.

    You may be prompted that your computer needs to be restarted. Safely shut down all programs and do so.

Using COM in VTS

This chapter will assist you in using the component object model (COM) in VTScada. Before you reading this section, please review and become familiar with the following definitions.

Automation Interface	A COM interface that uses late binding.
CLSID Class Identifier	A GUID that represents a COM class
COM Component Object Model	A software architecture that enables the components made by different software vendors to be combined into a variety of applications.
DDE	Direct Data Exchange.
GUID	A Globally Unique Identifier.
OLE	Object Linking and Embedding.
ProgID	Program Identifier. A human readable form of a class identifier.
SRO	Scope Resolution Operator or "\".
VTable Interface	A COM interface that uses early binding.

Related Information:

...Introduction to COM

...Accessing COM Objects

...Syntactic Structure

...Sample Code

...Functions and Statements Related to COM

Introduction to COM

This document assumes a basic knowledge of what the Component Object Model (COM) is and how it functions. You do not require an in-depth knowledge of COM to be able to use it effectively in VTScada.

Introductory information from Microsoft and others may be referred to if you require detailed background information on COM.

> Note: The Platform SDK documentation is provided with Visual C++, and is available for download from Microsoft. This provides a good starting point for those new to the Component Object Model.

A COM object is a piece of code that exists in an in-process DLL or out-of-process executable file on a computer. The COM object encapsulates a set of behaviors that can only be accessed via a set of formally declared "interfaces". An interface contains a list of member methods (along with their typed parameters), that can be called. All interface methods return a result code of type HRESULT. These interfaces can be classified into two types:
Virtual Table interfaces and Automation interfaces
These two interfaces are described as follows:

VTable Interfaces

VTable or "Virtual Table" interfaces are very closely related to the virtual function table that would be generated for indirectly calling C++ virtual functions. Automation interfaces are defined by Microsoft in their OC96 specification and, essentially, provide a limited set of strongly typed functions that can be called from C++ or other compatible languages. As those functions exist in a VTable interface, knowledge about the function names, calling conventions, and parameter types must be known when the compatible language is compiled. This is termed "early binding".

Automation Interfaces

The early-bound functions of an automation interface, however, provide a mechanism to access a much broader range of methods that the object supports. The calling convention and binding of names to physical functions is performed at run-time, and does not have to be known at compile time. The automation interface mechanism also provides dynamic discovery of parameter type information. Such discovery and subsequent usage of the discovered functions is termed "late binding". These

interfaces have largely grown out of the need for scripting languages to use COM objects without requiring compile-time knowledge of the object's methods.
VTScada uses the late binding ability of automation interfaces to interact with a COM object.

Accessing COM Objects

COM defines standard ways to instantiate an object, regardless of the location of the server code that generates the object instance. An object may be instantiated in-process by a DLL server, out-of-process on the same computer by an EXE server, or on another computer that is also an EXE server. Regardless of the object's location and the method of construction, the syntactic constructs to manipulate the object are identical. VTScada script code that constructs such object instances and manipulates them has no knowledge of the object's location.
A COM object may be identified by a "CLSID" (class ID), comprising a 38-character string that consists of an opening curly brace, a 36-character GUID (Globally Unique Identifier), and a trailing closing curly brace.

> Note: Each application has its own Globally Unique Identifier (GUID), which is generated by VTScada when the application is created. It can be found in the Information page of the Application Configuration dialog.

Internally, the operating system uses the GUID to look-up instantiation information for the specified object in the system registry. This notation is hardly human-friendly however, and so another translation exists, allowing an object to be identified by a "ProgID" (Program ID). The ProgID is then internally translated into a CLSID by the operating system.
A ProgID is a text string typically identifying a server code for associated objects, along with the identity of the object itself (for example, "TrihedralWidgetServer.Widget2" where "TrihedralWidgetServer" is the server code, and "Widget2" is the object).
The look-up information for such translations and the ultimate identification of the location of the object's server is all contained in the

system registry, as such information is normally programmatically stored there during the installation process for the object's server.
To create an instance of a COM object, you may use the COMClient statement. This statement takes a parameter representing the CLSID or ProgID of the object to be created (among other items), and assuming that this identifier does indeed map to a COM object, returns a value that may be assigned to a variable. This variable is an opaque handle, termed the "COM Client Interface" and is the only way to subsequently manipulate the object instance.
Manipulation of the object instance occurs via "properties" and "methods". You can think of properties as being the data that the object holds. The automation interface provides a "property get" ability, and a "property set" ability, to allow the properties contained within the object to be manipulated. The automation interface also provides the ability to call object methods.
The distinction between accessing a property and calling a method can sometimes be blurred. For example, it may be that the object implements a method that simply changes or returns the value of an internal property. Some properties may be parametrized; in other words, setting the property value requires more than one parameter to be passed. The syntactic structure of COM calls in VTScada however, conceals such confusions by providing a set of uniform object manipulation operations, where parameters passed to property manipulations are done in a manner identical to those in a method call.
When using a VTable interface (see Introduction to COM), the set of properties and methods, along with their parameter types, are normally expressed in source code form in a header file. However, automation interfaces had such metadata programmatically generated into a binary file called a "type library" when the object's server was built. The VTScada engine uses the type library's metadata to convert between VTScada values and the target object's expected value types, and to locate the correct property or method within the target object. The VTScada programmer can either use the object's formal documentation or one of

the various commercially available "type library browsers" to examine the metadata stored in a type library, and hence, discover the methods and properties that the object supports.
Each property get, property set or method call is described in the type library. Each parameter will be described not only by its type, but will also carry a directional specification such as [in] [out] or [in, out]. The directional specification indicates whether the parameter is an input to the instance ([in]), an output from the instance ([out]) or is both ([in, out]). By specifying a non-constant VTScada variable as a parameter for [out] and [in, out] parameters, the VTScada variable value will be updated at the conclusion of the property or method invocation.

Syntactic Structure

The implementation of COM in VTScada is intended to mimic the natural object model used by VTScada. As mentioned in Accessing COM Objects, opaque handle held in a variable instance that has a ValueType of "COM Client Interface" references an instance of a COM object. VTScada itself does not act as a server for COM objects; rather, it acts merely as a client. In other words, VTScada provides no COM objects; it simply provides the ability to use COM objects.
All properties and methods of the COM object are accessed using the scope resolution operator [SRO] "\". This presents the VTScada programmer with both syntactic and semantic compatibility with existing VTScada objects.
For example, a VTScada module can be instantiated, creating an object instance. The object instance can be scoped into to examine or modify values within the object. An object instance may provide methods that can be invoked. Only the instantiation syntax differs between the same operations, performed on a COM object. This is a necessary departure in order to describe the parameters necessary to locate an object server and cause it to instantiate an object.
Consider the following code:

```
[
MyVTSObject Module;
```

```
VTSObj;
COMObj;
]
Init [
If 1 Main;
[
{ Instantiate a VTScada object }
VTSObj = MyVTSObject();
{ Instantiate a COM object }
COMObj = COMClient("Trihedral.Widget");
]
]
Main [
]
```

Firstly, an instance of the VTScada module "MyVTSObject" is instantiated. The object value returned from the instantiation is stored in variable "VTSObj". Next, an instance of the COM object identified by the ProgID "Trihedral.Widget" is instantiated, and the COM Client Interface value is stored in variable called "COMObj".

Suppose that you wish to modify the contents of a value, held in the variable "MyValue" in the VTScada object created above. You could use code such as:

```
VTSObj\MyValue = 42;
```

Similarly, if the COM object instantiated in the above code had a property called "MyValue," you would use code such as:

```
COMObj\MyValue = 42;
```

This is termed a "property set" operation.

Reading the contents of a value in a VTScada object, or a property in a COM object is also similar:

```
VTSResult = VTSObj\MyValue;
COMResult = COMObj\MyValue;
```

For the COM object, this is termed a "property get" operation.

These operations can be combined:

```
VTSObj\MyValue2 += VTSObj\MyValue;
COMObj\MyValue2 += COMObj\MyValue;
```

Note that the normal expression operators [+= in this case] can be used just as easily with COM properties as with VTScada values.

An invocation of a method in a COM object is no different from invoking a method of a VTScada object:

```
VTSObj\MyValue2 = VTSObj\GetSomething(1, 2);
COMObj\MyValue2 = COMObj\GetSomething(1, 2);
```

In the first of the above two lines of code, a method contained in VTScada object instance is called with two numeric parameters. In the second, the same thing is happening with a COM object.
In the code above, the VTScada object was "launched". A VTScada object will remain running as long as its caller remains running. A COM object will remain running as long as there is a valid reference to it. VTScada objects can, however, be "called" from steady state. In this case, the VTScada object remains running until the state containing the call stops. Even if the VTScada object terminates, it will restart as long as the steady-state call remains running. A similar situation exists with COM objects. If a COM object is called from steady state, it will remain running as long as the steady-state statement remains running. A change of state will stop the COM object, even if there are other references held on it; in other words, the COM Client Interface value returned from the COMClient call is assigned elsewhere. Any variable value holding references to the COM object will be automatically invalidated when the COM object stops:

```
[
   MyVTSObject Module;
   VTSObj;
   COMObj;
   COMObjName;
]
Init [
   If 1 Main;
   [
      COMObjName = "Trihedral.Widget";
   ]
]
Main [
   { Instantiate a VTScada object }
   VTSObj = MyVTSObject();
   { Instantiate a COM object }
   COMObj = COMClient(COMObjName);
   If Trigger Done;
]
Done [
   { Both the VTScada object and the COM object are now stopped }
]
```

Like steady-state VTScada object calls, a steady state COM object call will re-trigger if its return value changes or any parameters to it change. In the case of a COM object, the return value would only change if some external event caused the COM object to be lost (e.g. communication failure, or if the parameter(s) to the COMClient statement caused the COM object to be destroyed). In the above code, changing the value of COMObjName would cause the existing COM object to be destroyed, the value of COMObj to be invalidated, and a new COM object to be constructed. Once the new object has been constructed, the value of COMObj will change to hold the COM Client Interface value for the new object.
Whether the object is a VTScada object or a COM object, properties and methods may also be accessed in both steady-state statements and scripts, but note that a property get in a steady-state statement will only evaluate once, as, unlike VTScada values, there is no automatic trigger from a COM object that a property value has changed. Instead, COM objects use "events" to indicate changes, and these events may call VTScada subroutines.
Consider the following code:

```
[
   Changed Module;
   COMObj;
   Latest;
]
Init [
   If 1 Main;
   [
      { Instantiate a COM object }
         COMObj = COMClient("Trihedral.Widget," Invalid, Self(),
Self(), Self());
   ]
]
Main [
   If Watch(Valid(Latest), Latest);
   [
      ...
   ]
]
<
Changed
(
   NewValue;
)
ChangedEvent [
   If watch(1);
```

```
   [
      Latest = NewValue;
      Return(0);
   ]
]
>
```

Note that the COMClient statement has grown four extra parameters. The first of these specifies the context in which it is permissible to instantiate the COM object. The second parameter specifies the scope in which event subroutines are to be found. All COM events have a name associated with them, defined by the COM object. If a subroutine module of the same name as the event exists in the scope specified in the third parameter to the COMClient statement, that subroutine will be called each time the corresponding event occurs. The subroutine is called on the same thread as the incoming event from the COM object, and so therefore may occur concurrently with other scripts running in the same scope.

The event subroutine is entered with whatever parameters were supplied by the COM object and the return value is passed back to the COM object as a result code (an HRESULT). Zero is a "success" return value. Each COM object defines what parameters are provided and what it expects the return value to be under different error conditions.

The event subroutine is run in the scope of the parent specified in the fourth parameter of the COMClient statement; therefore, any non-local variables referenced in the subroutine are resolved to values within that scope. In the above example, "Latest" is resolved to the value of "Latest" within the module that made the COMClient instantiation. The fifth parameter specifies the caller scope that is set up for the subroutine invocation. While this is normally meaningless for a subroutine, it can be used to pass "auxiliary" scope to the subroutine, adding flexibility.

The parent and caller parameters are optional, and if not specified, will default to Self(), Self(). The event subroutine search scope parameter is also optional; however, failure to specify a valid search scope, or subsequent invalidation of that parameter, prevents event subroutines from being called. By specifying a variable as the event subroutine search

scope parameter, you can enable and disable event subroutine calls, or even move the search scope for them.
The code above, then, uses a COM event to re-trigger the single statement in state Main when the COM object raises a "Changed" event that changes the value of "Latest". This is the primary method by which COM objects cause event-driven processing in VTScada.

Sample Code

In the example of the previous topic, the object that has been instantiated has been a hypothetical object. This section shows a simple working example that reads and writes spreadsheet cells in Microsoft Excel™. This example therefore, requires that Microsoft Excel be installed on your system.
Admittedly, this example doesn't do anything that couldn't be done using DDE, but it serves to illustrate come techniques when using COM interfaces.
Each of the main segments of code in this section can be concatenated together to make a complete script application that will compile and run.

```
{============================ System ============================}
{================================================================}
[
   Graphics Module { Contains user graphics };
   WinTitle = "Excel COM Tester" { Window title };
   System { Provides access to system library functions};
]
Main [
   Window( 0, 0 { Upper left corner },
          400, 160 { View area },
          400, 160 { Virtual area },
          Graphics() { Start user graphics },
           {65432109876543210}
          0b00010000000110011, WinTitle, System\DialogBGnd, 1);
]
<
{====================== System\Graphics ======================}
{ This module handles all of the graphics for the application }
{================================================================}
Graphics
[
   ExcelObj { The COM interface to the Excel object };
   ProgID { The ProgID for the Excel object };
   RangeObj { The COM interface to a "range of cells" object};
   CellsRead { Result of reading back some cells from Excel };
   CellsWritten { Cell values to be written to Excel };
```

```
   Kill = 0 { Gets set non-zero to terminate things };
   Row { Loop counter variable };
   Col { Loop counter variable };
   ValuesText { Text of the values read from spreadsheet };
]
Main [
{***** Instantiate the Excel object...Nothing happens yet *****}
   ExcelObj = COMClient(ProgID);
   {* But when the Excel object does become valid, make it visible
***}
   If Edge(Valid(ExcelObj), 1);
   [
      ExcelObj\Visible = 1;
   ]
   {***** When the Create button is pressed, make the ProgID valid.
This causes the COMClient statement, above, to retrigger and ExcelObj
to become valid *****}
   If ZButton(10, 30, 90, 10, "Create", 1, System\DefFont);
   [
      ProgID = "Excel.Application";
   ]
```

Initially, "ExcelObj" will be Invalid. When the "Create" button is pressed, the ProgID is made valid, the COMClient statement re-triggers, and the Excel object is instantiated. On instantiation, the Excel application is run (if not already running), but remains invisible. "ExcelObj" becoming valid causes the second statement to execute, which makes the Excel application window visible. Invisibility is not a general trait of such COM objects, but rather the behavior of Excel. It is possible to leave the Excel application invisible and still use it. You can try this by removing the property set:

```
ExcelObj\Visible = 1;
```

The completed application will function identically, but Excel will remain invisible.

The next requirement is code to shut down the COM object. Changing state would be sufficient to release the reference held on Excel and cause it to shutdown; however, Excel will not shutdown automatically if you have modified data within it, so you must make additional method calls to cause this to happen.

```
{***** Shutdown code. Pressing the "Kill" button stops the object
only. Hitting the toaster bar close stops the object and kills this
application *****}
If ZButton(110, 30, 190, 10, "Kill," 2, System\DefFont);
[
```

```
   Kill = 1;
]
If WindowClose(Self());
[
   Kill = 2;
]
If Kill;
[
   { Close down the Excel workbook and quit Excel }
   ExcelObj\Workbooks(1)\Close(0);
   ExcelObj\Quit();
   { Excel will run until the hold is released}
   ProgID = Invalid;
   { Terminate this application if so instructed }
   IfElse(Kill == 2,
      Slay(ParentObject(Self()), 0);
   { else }
      Kill = 0;
   );
]
```

In the above code, the variable "Kill" is set to "1" if the "Kill" button is pressed, and is set to "2" if the application is closed. To force Excel to close with modified data, the following two method calls are made on the Excel object:

```
ExcelObj\Workbooks(1)\Close(0);
ExcelObj\Quit();
```

Note that this would not be necessary if no changes were made to the data that Excel was operating with, and...

```
ProgID = Invalid;
```

...would be sufficient to close Excel. Similarly, a change of state would cause the Excel COM object to shut down. However, the simple application given here has only one state, with the COMClient statement being steady-state, so invalidating the ProgID or terminating the application is the only way to release the object.

The statement above:

```
ExcelObj\Workbooks(1)\Close(0);
```

...illustrates the use of "nested" interfaces. "ExcelObj" contains one or more "WorkBook" objects that can be accessed via the WorkBooks() method. For example:

```
ExcelObj\Workbooks(1);
```

returns a COM Client Interface to the first workbook. Because the value returned is a COM Client Interface, it can be used to call methods on the object to which it is connected. This can be done either by storing the COM Client Interface in a variable for later use:

```
WorkBookObj = ExcelObj\Workbooks(1);
WorkBookObj\Close(0);
```

or by a direct call:

```
ExcelObj\Workbooks(1)\Close(0);
```

In the latter case the "Workbooks" COM Client Interface will only exist temporarily – for the duration of statement execution.
The Excel object just instantiated, though, has no "workbook" object. To create one is simply another method call, which is made in response to the user pressing he "Workbook Add" button:

```
{***** Other buttons...Add a workbook *****}
If ZButton(10, 90, 90, 70, "Workbook Add," Valid(ExcelObj) ? 3 : 0,
System\DefFont);
[
   ExcelObj\Workbooks\Add();
]
```

Excel cells are represented by a "Range" object; an interface that can be obtained from the ExcelObj by a simple method call. This is made in response to the "Range Get" button:

```
{***** Get an object which represents a range of cells *****}
If ZButton(110, 90, 190, 70, "Range Get," Valid(ExcelObj) ? 4 : 0,
System\DefFont);
[
   RangeObj = ExcelObj\Range("A1:C1");
]
```

Assigning values to a range of cells is done by setting the "Value" property of the Range object to the VTScada values that will occupy those cells. Note that you can pass VTScada arrays as well as scalar values to a COM method or property. This is done in response to the user pressing the "Put A1:A3" button:

```
{***** Set the values of a range of cells *****}
If ZButton(10, 120, 90, 100, "Put A1:A3," Valid(ExcelObj) ? 5 : 0,
System\DefFont);
[
   CellsWritten = New(3, 1);
```

```
   CellsWritten[0][0] = 11;
   CellsWritten[1][0] = 12;
   CellsWritten[2][0] = 13;
   ExcelObj\Range("A1:A3")\Value = CellsWritten;
]
```

Similarly, reading back a range of cells is done through the same "Value" property. However, this time the property is used on the right-hand side of the expression, and so is an implicit "property get," rather than the implicit "property put" that resulted from using the property on the left-hand side, above:

```
{***** Get the values of a range of cells *****}
If ZButton(110, 120, 190, 100, "Get A1:A3," Valid(ExcelObj) ? 6 : 0,
System\DefFont);
[
   CellsRead = ExcelObj\Range("A1:A3")\Value;
]
```

Excel requires that the data supplied to the "Value" property put is dimensioned exactly the same as the range of cells into which the data is being put. Hence, the array dimension used was 3 rows by 1 column. Exactly the same holds true for putting data into multiple columns of the same row. This example places 1 row of 3 columns into the workbook:

```
{***** Set the values of a range of cells *****}
If ZButton(10, 150, 90, 130, "Put B1:D1," Valid(ExcelObj) ? 7 : 0,
System\DefFont);
[
   CellsWritten = New(1, 3);
   CellsWritten[0][0] = 21;
   CellsWritten[0][1] = 22;
   CellsWritten[0][2] = 23;
   ExcelObj\Range("B1:D1")\Value = CellsWritten;
]
{***** Get the values of a range of cells *****}
If ZButton(110, 150, 190, 130, "Get B1:D1," Valid(ExcelObj) ? 8 : 0,
System\DefFont);
[
   CellsRead = ExcelObj\Range("B1:D1")\Value;
]
```

In both cases, the result of the property get is an array dimensioned exactly the same as the cell dimensions.

The example concludes with code to display the results. The first two ZText statements will both display the COM Client Interface values obtained above, which will be displayed as a text string representing the ProgID of the interface:

```
{***** Display code, to show what is going on *****}
ZText(210, 27, Concat("Interface Value: ," PickValid(ExcelObj,
"Invalid")), 0, System\DefFont);
ZText(210, 87, Concat("Range Value: ," PickValid(RangeObj,
"Invalid")), 0, System\DefFont);

Note that the Range interface appears to have a ProgID, even though
it is not a "creatable" interface (i.e. you could not use it as a Pro-
gID in a COMClient statement). Strictly speaking, what you see is a
textual representation of the COM Client Interface name.
```

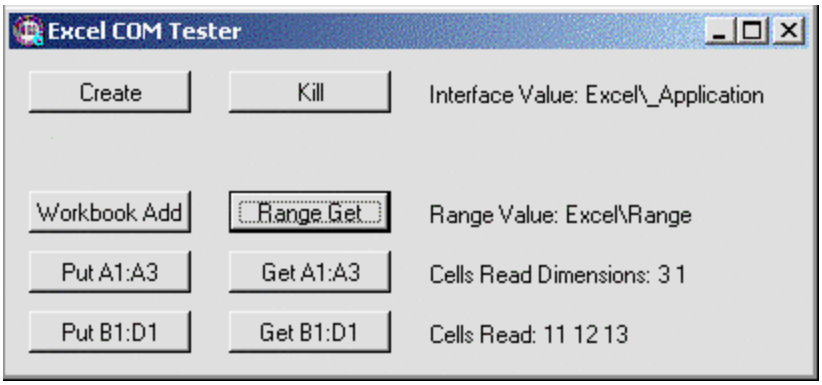
```
The next statement displays the dimensions of the array read back
from a range of cells.
```

```
ZText(210, 117, Concat("Cells Read Dimensions: ," PickValid(ArraySize
(CellsRead, 0), "Invalid"),
      " ," PickValid(ArraySize(CellsRead, 1), "Invalid")),
      0, System\DefFont);
```

Finally, the last code segment renders the values read from the cells into a text string and displays it:

```
If Watch(1, CellsRead);
[
   Row = 0;
   ValuesText = "Cells Read: ";
   WhileLoop(Row < ArraySize(CellsRead, 0),
      Col = 0;
      WhileLoop(Col < ArraySize(CellsRead, 1),
         ValuesText = Concat(ValuesText, CellsRead[Row][Col], " ");
         Col++;
      );
      Row++;
   );
]
ZText(210, 147, ValuesText, 0, System\DefFont);
]
{ End of System\Graphics }
>
```

Compiling and running the application will result in the following user interface display:



Functions and Statements Related to COM

The following functions are related to COM usage in VTScada.

...ActiveX - Instantiates an ActiveX object.

... COMClient - Instantiates COM objects that do not possess a user interface.

... COMEvent - Sets an event subroutine context for an existing COM client interface.

... COMPort - Opens a serial port and handles all interrupts and asynchronous events for that port.

... COMStatus - Returns the last status information that occurred for a specified COM client interface.

Using DDE

DDE (Dynamic Data Exchange) is a mechanism within Windows that enables data to be exchanged between two different programs, such as VTScada and Microsoft Excel. The program that supplies the data is called the server, while the program that accepts the data is referred to as the client. VTScada can act as either a client or a server.

Related Information:

See the VTScada Developer's Gide for:

...VTScada as a DDE Client

TCP/IP Networking

TCP/IP support is integrated into VTScada. To use this feature you will need to have a TCP/IP stack, which is supplied by many network vendors. All TCP/IP functions in VTScada are performed using socket streams that act like serial connections between two programs. VTScada can act as both a socket server and a socket client.
Following is an example of a Client and a Server that will create a connection and pass the string "Hello World" followed by a number representing the number of seconds since midnight.

Client:

```
[
   Graphics Module { Contains user graphics };
   Calculations Module { Contains user calculations };
   WinTitle = "Socket Test - Client Side" { Window title };
   SocketHandle;
   Client;
   Server;
   Data;
]
Main [
   Window( 0, 0 { Upper left corner },
           800, 600 { View area },
           800, 600 { Virtual area },
           Graphics() { Start user graphics },
          {5432109876543210}
         0b0010000000110011, WinTitle, 0, 1);
]
<
{==================== System\Graphics ======================}
{ This module handles all of the graphics for the application }
{===========================================================}
Graphics
Init [
   If 1 Screen1;
   [
      Client = ClientSocket(0, "Richard," 20000, 1024, 1024, 1);
   ]
]
Screen1 [
   If TimeOut(!Valid(Client), 2) Init;
   If TimeOut(ValueType(Client) <> 8,2) Error;
```

```
   If GetStreamLength(Client) > 0 || MatchKeys(2, "r");
   [
      SRead(Client,
      Concat("%," Concat(GetStreamLength(Client), "c")), Data);
   ]
   If TimeOut(1, 1);
   [
      SWrite(Client, "%s," Concat(" Hello World ," Time(Seconds(),
3)));
   ]
   If MatchKeys(1, " ");
   [
      SWrite(Client, "%s," Concat(" Hello World ," Seconds()));
   ]
   ZText(10, 150, Data, 15, 0);
   ZText(200, 100, Cond(Valid(Client),"Connected,""Not
Connected"), 10, 0);
   ZText(200, 110, Concat("ErrorCode : ," Client), 10, 0);
   ZText(200, 120, Concat("Type : ," ValueType(Client)), 10, 0);
   If WindowClose(Self());
   [
      CloseStream(Client);
      Slay(Self(), 1);
   ]
]
Error [
   ZText(100, 130, Concat("ErrorCode : ," Client), 10, 0);
]
{ End of System\Graphics }
>
```

Server:

```
[
   Graphics Module { Contains user graphics };
   Calculations Module { Contains user calculations };
   WinTitle = "Socket Test - Server Side" { Window title };
   SocketHandle;
   Client;
   Server;
   Data;
   Attribs0;
   Attribs1;
]
Main [
   Window(0, 0 { Upper left corner },
          800, 600 { View area },
          800, 600 { Virtual area },
          Graphics() { Start user graphics },
         {5432109876543210}
        0b0010000000110011, WinTitle, 0, 1);
]
<
{==================== System\Graphics ======================}
{ This module handles all of the graphics for the application }
{===========================================================}
Graphics
```

```
Init [
   If 1 Wait ;
   [
      SocketHandle = SocketServerStart(0, 20000, 1024, 1024, 1);
   ]
]
Wait [
   If SocketWait(SocketHandle) Main;
   [
      Server = ServerSocket(SocketHandle);
   ]
]
Main [
   If GetStreamLength(Server) > 0 || MatchKeys(2, "r");
   [
      SRead(Server, Concat("%," Concat(GetStreamLength(Server),
"c")), Data);
      SWrite(Server, "%s," Data);
   ]
   If WindowClose(Self);
   [
      CloseStream(Server);
      SocketServerEnd(SocketHandle);
      Slay(Self(), 1);
   ]
   ZText(0, 50, Data, 15, 0);
   ZText(0, 100, Cond(Valid(Server), "Connected,""Not Connected"),
10, 0);
]
{ End of System\Graphics }
>
```

Related Functions:

The main functions used to handle TCP/IP are as follows:

... BlockWrite

... ClientSocket

... ServerSocket

... SocketServerStart

... SocketServerEnd

... SocketWait

... SRead

... SWrite

... TCPIPReset

SNMP Agent Configuration

The SNMP agent is not enabled by default. You must enable and configure the agent before using. All of the application properties listed at the end of this topic should be reviewed.
Once configured, you will be able to serve tag values over SNMP. An SNMP client will be able to connect to the server to read from or write to tags configured for SNMP access. The following features are available:

- Access to VTScada tag values via SNMP GetRequest commands.
- Ability to set VTScada tag values via SNMP SetRequest commands.
- Ability, using custom code, to send Trap/InformRequest notification to the NMS[1] when tag values change.
- Support multiple simultaneous client connections.
- The address (OID[2]) assigned to a tag is retained and existing address bindings do not change when regenerating or updating the MIB[3].
- You can export the SNMP Agent configuration as a MIB file, allowing VTScada settings to be imported into a 3rd party system.
- The enterprise PEN may be customized.
- The community names are configurable.

Warning: Do not allow write access over an unsecured network. Community strings are merely plain-text passwords.

To enable the SNMP agent:

1. Open the Edit Properties page within the Application Configuration dialog.
2. Select the Advanced Mode option.
3. Locate the property SNMPAgentEnable.
   This will be an OEM property, unless previously configured in your applic-

[1]Network Management System

[2]Object IDentifier. A part of the SNMP driver addressing system.

[3]Management Information Base. A hierarchy of the information available to an SNMP device, organized by numbered Object Identifiers (OIDs).

ation. If so, you must copy the property to your application before you can change the value.

4. Set the value of the local copy of SNMPAgentEnable to 1.
5. Save and apply changes.

The SNMP Agent requires an IP Network Listener tag to provide access to the system. That tag provides features such as IP filtering and connection audit logs. The IP Network Listener tag must be configured with the name specified in the application property, SNMPAgentIPListener .

1. Locate the property, SNMPAgentIPListener.
2. Make note of the value of that property.
3. Ensure that you have an IP Network Listener tag located at the top level of the tag hierarchy, and having a name that is identical to the value shown in the property, SNMPAgentIPListener.

Related Information:

...MIB Objects

...Agent Tag Setup

...Agent Tag Fields

...Trihedral MIB Definition

...Agent Tag Change Notification Traps

...Custom MIB Setup

...Support for Analog Tag Values

...Support for Data Time Stamps

Refer to the VTScada Admin Guide for:
Service enabling properties:

...SNMPAgentEnable

...SNMPAgentReadCommunity

...SNMPAgentWriteCommunity

...SNMPAgentWriteEnable

...SNMPAgentIPListener

Advanced communication properties:

...SNMPAgentMaxTCPSize

...SNMPAgentMaxUDPSize

...SNMPAgentSessionTimeout

Trap configuration properties:

...SNMPAgentTrapCommunity

...SNMPAgentTrapHost

...SNMPAgentTrapPort

...SNMPAgentTagNotifyMode

Properties used only to inform traps:

...SNMPAgentInformRetryInterval

...SNMPAgentInformRetryLimit

MIB Objects

The VTScada SNMP Agent implements MIB-II, which is a basic object available on most SNMP agent devices. (See: RFC 1907 - Management Information Base for SNMPv2)

This provides certain basic objects such as the device identification and message processor success and error counts. VTScada fully implements MIB-II as per compliance requirements.

Custom objects implemented by the VTScada SNMP Agent are located under the vtscada OID (1.3.6.1.4.1.42905.1). The top level nodes of the product are as follows:

appTraps(0)	definitions for custom trap types (not accessible)
appInfo(1)	application server information such as application name and memory usage
appNotifyInfo(2)	Notification subsystem information variables
appTags(3)	Contains all the tags configured for use with the SNMP Agent

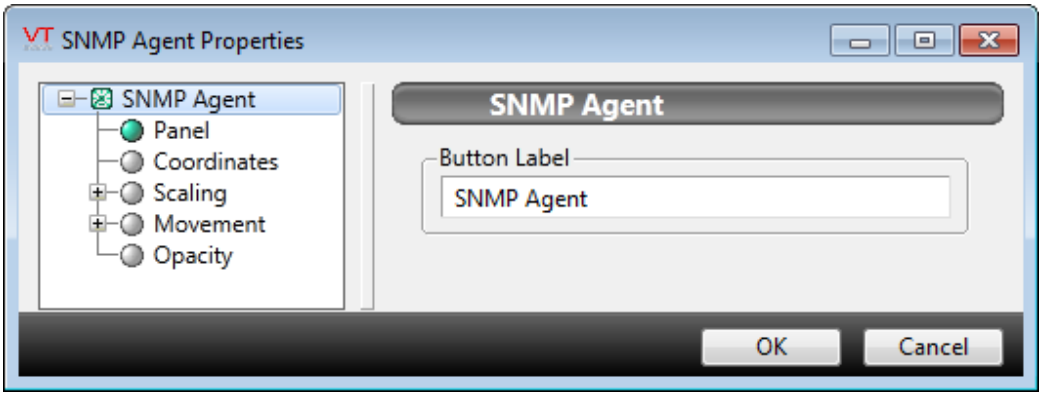
Agent Tag Setup

By default there will be no tags available in the agent MIB, only basic service variables. Tag setup requires a manual action to initiate. In

particular, the property, SNMPAgentEnable must be set to 1 before proceeding.

1. Open the Idea Studio.
2. Within the Widgets palette, open the Tools folder.
3. Open the SNMP Agent Tools folder
4. Add the 'SNMP Agent' button to your page.

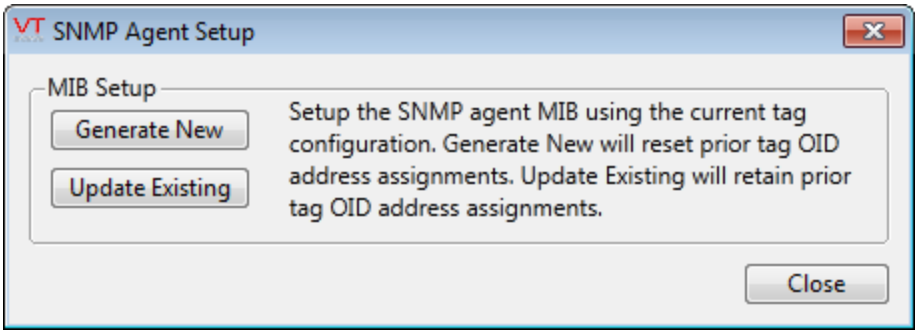
The SNMP Agent button has only one configurable property: the name to be displayed on the button.



5. Close the Idea Studio.

Running the SNMP Agent:

1. Click the SNMP button.
   The SNMP Agent Setup dialog will open.



There are two options: Generate New and Update Existing.

a. Generate New will reset prior OID address assignments
b. Update Existing will retain prior OID address assignments
   (Either way the SNMP Agent will be updated to the current VTScada tag configuration)

Select the desired option and then choose a location in which to save the exported MIB file.
Only OPC-enabled tags can be made available via the SNMP Agent. The update process automatically includes all OPC-enabled tags running in the application.

- OID addresses are assigned to tags and are retained until explicitly reset.
- Every available tag will have the following fields: { Name, Value, Quality, Timestamp, Type }
- SNMP data type and access level for Value are determined via the OPCGetTagAttributes() API.
- A subset of the values from OPCGetTagProperties() populate the optional fields: { Description, Area, Units, Device, Address }
- SNMP Get commands use the OPCReadTagValue() API to access the Value, Quality, and Timestamp.
- SNMP Set commands use the OPCWriteTagValue() API to set the Value.

See the OPC setup guide for OPC API function implementation.

Note: subsequent tag configuration changes will not be included until the next time MIB Setup is executed.

Related Information:

...Agent Tag Fields

Agent Tag Fields

The following set of fields are potentially available for every OPC-enabled tag:

- **Name** – The full VTScada tag name
- **Value** – Value if the tag if Quality is good
- **Quality** – quality of the tag Value, as follows:
  - Bad (0)
  - Configuration Error (4)
  - Uncertain (64)
  - Engineering units exceed low limit (85)
  - Engineering units exceed high limit (86)

    - Good (192)
    - Good with manual override (216)
- **Timestamp** – UTC timestamp associated with the tag
- **Type** – Name of the tag type
- **Description** – Description of the tag
- **Area** – Area of the tag
- **Units** – Engineering units associated with the tag value
- **Device** – Name of the associated I/O device tag
- **Address** – I/O device address associated with this tag

Except for Value, all fields are read-only.
Quality values 85 and 86 will occur only for VTScada tags that map to 32-bit integers, such as the Counter tag.

Trihedral MIB Definition

The exported VTScada SNMP Agent MIB is defined under the Trihedral enterprise. Most management software will require the Trihedral MIB definition in order to process the VTScada MIB.
The Trihedral enterprise MIB definition:

```
TRIHEDRAL-MIB DEFINITIONS ::= BEGIN

IMPORTS
         MODULE-IDENTITY, enterprises FROM SNMPv2-SMI;

trihedral MODULE-IDENTITY
    LAST-UPDATED "201312020000Z"
    ORGANIZATION "www.trihedral.com"
    CONTACT-INFO ""
    DESCRIPTION "Trihedral Engineering Ltd."
    REVISION "201312020000Z"
    DESCRIPTION "First draft"
    ::= { enterprises 42905 }

END
```

A copy of this definition is present in the installation directory under Mib-s\TRIHEDRAL-MIB.txt

Agent Tag Change Notification Traps

The SNMP Agent provides a custom trap type for sending tag change notifications to the management server. Custom code is required to send a change notification, but this can be done from a simple child tag that watches the parent value, or from an application service module. This provides maximum flexibility as to when and how the notifications are sent.

\SNMPAgent\NotifyTagChange

Description: Sends a notifyTagChange trap to the configured trap destination.

Returns:

Usage: Subroutine

Format: SNMPAgent\NotifyTagChange(TagName[, NeedAck, Fields])

Parameters:

TagName

Required. Name of the tag to send the notification for

NeedAck

Optional. Any Boolean expression. Set TRUE if acknowledge is required or FALSE to send unacknowledged trap. Defaults to FALSE.

Fields

Optional. An array of field names to send in trap/inform PDU. Defaults to (Timestamp, Value, Quality). See available field names under Agent Tag Fields.

Comments: This sends a notifyTagChange trap to the configured trap destination. By default the SNMP Agent notification subsystem will only send notification from the current server.

The option SNMPAgentTagNotifyMode can adjust this behavior as needed. The name of the RPC service for notifications is "SNMP Agent Notifications", which can be configured like any other VTScada RPC service.

If the SNMP Agent Notifications service server changes then a notifyServerChange trap will be sent from the new server. The management system can choose to poll for the current tag values whenever this occurs, as notifications may have been missed during the transition period.

The following custom appNotifyInfo MIB objects also provide some useful information.

- notifyServer(1) – 1 if this is the current notification server
- notifyFailCount(2) – count of notifications that did not receive an acknowledge (after retries).
- lastNotifyFailUTC(3) – UTC timestamp of the last notifyFailCount

A management system relying on *notifyTagChange* notifications to keep values updated should keep an eye on the fail statistics, polling for current tag values whenever synchronization has been lost as indicated by an increase in the fail count.

Custom MIB Setup

The option exists to configure the SNMP Agent MIB to produce an OEM MIB rather than the standard VTSCADA MIB. The primary setting to enable this is:

SNMPAgentOEMSetup
This is a string of the form "enterprise(PEN).product(number)". (For example: "acme(696).rocketApp(1)").

The enterprise and PEN must match what is defined in the enterprise MIB file. Once this setting is specified you must include the following in order to generate a functional MIB file:

SNMPAgentOEMEnterpriseMIBModule
Enterprise MIB module name for exported MIB file (Eg; "ACME-MIB"). Must match the enterprise MIB file module declaration.
SNMPAgentOEMProductMIBModule
Product MIB module name for exported MIB file (Eg, "ROCKET-APP-MIB")

The MIB subsystem will create a custom MIB based on these settings. Note that the enterprise MIB file will have to be supplied to the end user in order for the custom product MIB to be of use. However, the VTScada SNMP Agent does not need the actual enterprise MIB module to function. The header of the generated MIB file will contain the following fields. Definitions for these are optional.

SNMPAgentOEMOrganization
(string) Organization for exported MIB file

SNMPAgentOEMContactInfo
(string) Contact Information for exported MIB file

SNMPAgentOEMDescription
(string) Product description for exported MIB file

In order for the custom parameters to be SMIv2 compliant, the following rules must be observed,

A. MIB Module names:
- Must begin with UPPER case letter; followed by zero or more letters, digits, and hyphens
- Hyphen cannot be last, and there cannot be two consecutive hyphens

B. SNMPAgentOEMProduct setting names:
- Must begin with LOWER case letter; followed by zero or more letters and digits
- Have no hyphens

Support for Analog Tag Values

Standard SNMP does not define a floating point data type. To provide this values over SNMP, without truncating to integers, the floating point numbers are encoded as strings by default.
For example, 1234.567 is returned as a DisplayString with value "1234.567".
Tags with integer values are encouraged to report their type as integer rather than a floating point to avoid such issues.

Related Information:

...Support for Data Time Stamps

Support for Data Time Stamps

The timestamps (type: timeticks) provided by SNMP are based on sysUpTime (see MIB-II), which is relative to the device restart rather than any useful epoch. Further, sysUpTime will roll over eventually.
To provide for tag values, and other useful timestamps, the SNMP Agent encodes timestamps in type Unsigned32 as the number of seconds since midnight of January 1, 1970 (where "midnight" is 00:00). By convention all timestamps will be in UTC.

Related Information:

...Support for Analog Tag Values

Using ODBC

ODBC (Open Data Base Connectivity) is a standard interface to a wide variety of database packages. ODBC drivers are available for most common databases. With the ODBC interface, a VTScada application can be written to store and retrieve data from the database in a format independent of the particular database used. To switch databases, only the driver needs to be changed.

Note: In order to perform ODBC operations, an ODBC driver for the DBMS that you wish to use must be installed on your system and properly configured. Consult your DBMS vendor documentation for details of how to do this.

The program, VTSODBCDriverInstall.EXE (distributed with VTS), will install an OBDC driver on your system. This driver is compatible with both 32 and 64 bit Windows versions. The VTScada OBDC driver will allow you to access VTScada tag and alarm history data using SQL queries – subject to this option being purchased with your VTScada license.
To access an ODBC data source, a Data Source Name (DSN) is used. This can be configured using the Microsoft Windows ODBC Data Source Administrator (installed under Control Panel\Administrative tools) or (better) using a file DSN or "DSN-less" connection string.
32-bit ODBC data sources on a 64-bit Windows use the Microsoft Windows ODBC Data Source Administrator located in %systemroot%\SYSWOW64\odbcad32.exe. Consult the online Microsoft documentation for more details.

Related Information:

...SQL Queries of VTS Data – The ODBC Server – See the VTScada Developer's Guide

Using DLLs

DLLs are dynamic link libraries. They are the primary means for accessing C code functions from VTScada. With the DLL statement, you can access existing DLLs or access your own DLL used to create some specialty function. To create a DLL you will need a programming language compiler for a language such as C.

Related Functions:

... DLL

... LoadDLL

Modem Manager Service

The VTScada Modem Manager provides data and voice telephony services for standard VTScada applications.

Note: TAPI - Telephony Application Programming Interface. A component of Microsoft operating systems. The TAPI interface enables the connection of a PC running Windows to telephone services. The TAPI standard supports connections by individual PCs, as well as LAN connections serving many computers. Within each connection type, TAPI defines standards for simple call control and manipulating call content.

The Modem Manager provides services that allow modems connected to different machines to be managed as a common pool.
Key features of the Modem Manager are:

- Able to control a pool of modems distributed across a number of PCs. Any combination of your modems may be included in or excluded from a given pool. Within a pool, each modem is tried in turn until a call is successful.
- One more modems can be set aside from the pool, to be used as a preferred route for outgoing calls.
- Provides control over which modems should be used, and when calls should be made.
- Call setup, queue information, and progress information are distributed to all copies of Modem Manager on a networked system.
- Queues and timekeeping use UTC for coordination of modems across multiple time zones.
- Provides generic audio call handling facilities, including guard tone, DTMF detection, and speech synthesis.
- Provides a way for you to list the modems in your system.
- Trihedral provides both a TAPI Service Provider (TSP) and a modem audio driver. These have been developed to overcome limitations that were found in the standard Unimodem V driver from Microsoft.

The Modem Manager interfaces with a Microsoft system component called "TAPI" (see definition box). TAPI enables different modems to be

handled in a generic manner, and to be shared between different applications and services (e.g. RAS or FAX). TAPI does not interface directly to the modem; this is the job of the Trihedral Voice Modem Service Provider. You may use the more common Unimodem V, but a Trihedral driver was developed to avoid several problems that have been found with the Unimodem V driver.
Physical modems are associated with modem tags. These hold the configuration details for each modem, and may be monitored for call progress or errors.

Related Information:

...Modem Tags - Configuration of, described in the VTScada Developer's Guide.

...Modem Manager Concepts - An overview of what the Modem Manager does.

...Canonical Address Format - Reference: how to format telephone numbers.

...Modem Manager Configuration Variables - Control over modem functionality.

...Sequence of Events for Incoming Calls - How the Modem Manager works.

...Sequence of Events for Outgoing Calls -

...Allocating Modems in a Managed Pool for Outgoing Calls - Modem allocation.

... Local Modems

...Modem Manager Alarm and Event Reporting - Event codes

... Modem Manager API - Create a custom driver to handle calls.

...TAPI and UniModem Considerations - Common questions when configuring modems.

Modem Manager Concepts

Client Server Relationships

The Modem Manager runs as a server on one system and as a client on others. The server and client Modem Managers cooperate to provide modem pool management and to set up data paths between modem users and the allocated modem(s).

Outgoing Calls

When an outgoing call is requested, the Modem Manager selects a free modem, or queues the call until a modem becomes free. Once a modem is free, the call is dialed. The selected modem may be on a different PC than that requesting the call. The Modem Manager sets up transparent data paths between the modem and the requestor. In the event that the Modem Manager fails to complete the connection, it can retry the attempt using an alternative modem or modems, if they are available.

Incoming Calls

When an incoming call is received by one of the pool modems, the Modem Manager passes the initial data to objects that have previously registered an interest in incoming calls (discriminator modules).

- If an object accepts the offered call, then the Modem Manager sets up transparent data paths between the modem and the recipient.
- If no one accepts the call, then the Modem Manager returns the call to the operating system to allow other applications the opportunity to take the call.

This incoming discrimination can also function with incoming audio calls.

At any time there are three logical PCs involved in a call:

- The caller or receiver.
- The Modem Manager server.
- The owner of the physical modem.

Depending upon the configuration of your system, the roles of these three logical PCs may be played out on one, two, or three physical PCs.

Canonical Address Format

If VTScada is to use the system's modem dialing rules then the phone number to be dialed must be given in canonical format.
The canonical address format is intended to be a universally constant directory number; for this reason, numbers in address books are best stored using canonical format.
A canonical phone address is a text string with the following structure:

```
+ CountryCode SPACE [(AreaCode) SPACE] SubscriberNumber | Subaddress
^ Name CRLF …
```

The components of this structure:

Component	Meaning
+	Equivalent to hex 2B. Indicates that the number that follows it uses the canonical format.
CountryCode	A variably-sized string containing one or more of the digits "0" through "9" (hex 30 through 39 inclusive). The CountryCode is delimited by the SPACE that follows it. It identifies the country/region in which the address is located.
SPACE	Exactly one space character (hex 20). It is used to delimit the end of the CountryCode portion of the address.
AreaCode	A variably-sized string containing zero or more of the digits "0" through "9" (hex 30 through 39 inclusive). AreaCode is the area code portion of the address, and is optional. If the area code is present, it must be preceded by exactly one left parenthesis character (28), and be followed by exactly one right parenthesis character (29) and one space character (20).

<table>
<tr><td>Sub-
scriber-
Number</td><td>A variably-sized string containing one or more of the digits "0" through "9" (hex 30 through 39, inclusive). It may include other formatting characters as well, including any of the dialing control characters described in the Dialable Address Format
<table>
<tr><th>Character</th><th>Hex Encoding</th></tr>
<tr><td>!</td><td>20</td></tr>
<tr><td></td><td>23</td></tr>
<tr><td>$</td><td>24</td></tr>
<tr><td>*</td><td>2A</td></tr>
<tr><td>,</td><td>2C</td></tr>
<tr><td>?</td><td>3F</td></tr>
<tr><td>@</td><td>40</td></tr>
<tr><td>ABCD</td><td>41-44</td></tr>
<tr><td>P</td><td>50</td></tr>
<tr><td>T</td><td>54</td></tr>
<tr><td>W</td><td>77</td></tr>
<tr><td>abcd</td><td>61-64</td></tr>
<tr><td>p</td><td>70</td></tr>
<tr><td>t</td><td>4</td></tr>
<tr><td>w</td><td>9</td></tr>
</table>
The subscriber number should not contain the left parenthesis or right parenthesis character (these are used only to delimit the area code), nor should it contain the pipe (|), carat (^), or CRLF characters (which are used to begin following fields).
Most commonly, non-digit characters in the subscriber number would include only spaces, periods (.), and dashes (-). Any allowable non-digit characters that appear in the subscriber number are omitted from the DialableString returned by the LineTranslateAddress function, but are retained in the DisplayableString.</td></tr>
<tr><td>|</td><td>Hex (7C). If this optional character is present, the information following it up to the next + | ^ CRLF, or the end of the canonical address string, is treated as sub address information, as for an</td></tr>
</table>

	ISDN sub address.
Subaddress	A variably-sized string containing a sub address. The string is delimited by + | ^ CRLF or the end of the address string. During dialing, sub address information is passed to the remote party. It can be such things as an ISDN sub address or an email address.
^	Hex (5E). If this optional character is present, the information following it up to the next CRLF or the end of the canonical address string is treated as an ISDN name.
Name	A variably-sized string treated as name information. Name is delimited by CRLF or the end of the canonical address string and can contain other delimiters. During dialing, name information is passed to the remote party.
CRLF	Hex (0D) followed by Hex (0A), and is optional. If present, it indicates that another canonical number is following this one. It is used to separate multiple canonical addresses as part of a single address string (inverse multiplexing).

For example, the canonical representation of the main switchboard telephone number at Trihedral is:

```
+1 (902) 835-1575
```

Modem Manager Configuration Variables

Through the following application properties, you have extensive control over how the modem manager operates.
The following can be found in the VTScada Admin Guide:

...AnswerCalls

...CallInterval1

...CallOutDelay1

...CallOutDelay2

...CallOutPriority

...CycleDelay

...CycleLength

...DataIdleTime

...DialerSpeechInit

...DialResetTime

...DialWaitTime

...GuardTone

...HangUpDelay

...HelloPacketLength

...InitialDataDelay

...InitModemsDisabled

...MaxHandOffCount

...MinModemsFree

...MMCycleTime

...MMLogDateFormat

...MMLogLevel

...MMLogTimeFormat

...MMMaxQTime

...MMRPCTimeout

...ModemAlarm

...ModemAutoReset

...ModemManagerLogSize

...ModemMmaster

...<ModemName>Device

...<ModemName>Disabled

...ModemRetries

...ModemSpeechTO

...ModemTCPIPPort

...SiteRetries

...SquelchDetectDelay

...SquelchIdleTime

...SquelchPacketLength

...UseSerialAreaInModemCall

...UseUnimodem

Sequence of Events for Incoming Calls

The exact sequence of events depends on whether the modem is operating in data mode or in audio mode. This, in turn, is decided by the collective modem media modes specified by the active call discriminator modules.

Related Information:

...Modem in Data Mode

...Modem in Audio Mode

...Sequence of Events for Outgoing Calls

Modem in Data Mode

Note: The following italicized words are application properties. Information on each of these may be found in "Application Properties for the Modem Manager".

When a call is answered, the Modem Manager starts a configurable noise suppression phase as follows:

1. Wait for SquelchDetectDelay seconds, then go to step 3. If while waiting, data is received, then go immediately to step 2.
2. Throw away data until either:
    a. An idle period equal to SquelchIdleTime seconds occurs, in which case go to step 3; or
    b. Data received exceeds SquelchPacketLength, in which case the call is disconnected.
3. Wait for data to start arriving. When it does, go to step 4. If the wait exceeds InitialDataDelay seconds, disconnect the call. Alternatively, if

InitialDataDelay1 is specified, go to step 4 after the defined period of time. This allows for RTUs that don't speak until they are spoken to.

4. Every DataIdleTime seconds, monitor the total amount of data received in this step. If it is at least HelloPacketLength bytes, or does not change between two successive DataIdleTime periods, go to step 5.
5. Offer the received data packet to each registered driver.
6. The data is offered to the registered driver(s) by calling the module discriminator in the driver's scope. The data received so far is passed in a buffer as a parameter to the discriminator module. This subroutine examines the data and returns Invalid if the data is not recognized, or the station identifier of the driver instance that should handle the call. If the call is accepted, then Modem Manager will call the driver's Connect() module, passing the station identifier and the stream as parameters.

Note: An example of a data discriminator can be found in Example Data Discriminator.

Factors to Consider for the Configuration of Incoming Calls

- Will incoming calls be answered (AnswerCalls)?
- Set up modems (on a per modem basis) to answer on a specific number of rings (see: Modem Tag Type Properties: Settings Tab).
- Decide how many modems should be kept free to accept incoming calls (MinModemsFree).
- Specify noise filtering conditions to ensure that good connections are appropriately detected.
- Register the driver (provide a module) (see Modem Manager Programming Interface).
- Decide whether to hand-off unaccepted calls (MaxHandOffCount).

Modem in Audio Mode

An audio discriminator recites voice prompts to the caller, and determines the validity of the caller by received DTMF tones.
When a tag registers an audio discriminator object, that tag should have a DataPort variable and a module called AudioDiscriminator().

When a call is answered, the modem is initially in audio mode. The highest priority AudioDiscriminator() module is called with the tag's DataPort variable set to a useable stream.

- This is a steady state call that times-out after the period of time specified when the discriminator module was registered.
- If the discriminator module returns "0", the call is passed to the next discriminator module.
- If the discriminator module returns a non-zero result, then it is deemed to have accepted the call.

The discriminator may read and write on the DataPort stream.
Any data written to the stream is converted to speech using the voice identified when the discriminator was registered. The speech device is initialized using the string defined by the DialerSpeechInit application property. Data written may include any escape sequences meaningful to the TTS engine.
If a bookmark is set, when the bookmark is reported back to the Modem Manager, a single character equal to 0x45+Bookmark Number is be inserted into the stream, and may be read by the discriminator module.

- The only other data that can be received are DTMF tones inputted at the remote device (in order to receive such tones, this requirement must be specified in the media mode for the discriminator). This data appears in the stream as the digits 0..9, the number-hatch character (#), or the asterisk character (*).
- If the discriminator accepts the call, then DataPort remains as a valid stream, and the tag that owns the discriminator now has control.
- If the discriminator rejects the call or times out, then the call is passed to the next audio discriminator in priority order. If there are no more audio discriminators, the modem is switched into data mode, and the identification of the call continues as described in Modem in Data Mode (as if a data call had just been received).

Careful consideration needs to be given to the overall time tolerance of this sequence. If, say, a RAS call is received in audio mode, then it must go through all audio discriminators, the switching of the modem to data mode (which typically takes 10 seconds), and all the data discriminators,

before the call is handed off to RAS. This is not a deficiency in the Modem Manager, but a necessary consequence of accepting mixed media calls.

Note: An example of an audio discriminator can be found in Example Audio Discriminator.

Factors to Consider for the Configuration of Outgoing Calls
Local telecommunications authorities may have regulations regarding the frequency at which call attempts are made to a particular number. By defining values for the following modem-related application properties, you can set restrictions on redial attempts.

- CycleLength defines the number of steps in the cycle (with a maximum of 10),
- CallInterval1 through to CallInterval10 define the delay (in seconds) at each step transition.
- CycleDelay defines the final delay (in seconds) before the cycle restarts.
- HangUpDelay indicates the number of seconds to wait before hanging up the modem when there are not active attempts to read or write.
- DialWaitTime enables you to configure the number of seconds to wait before retrying a failed modem operation after no dial tone or response from the modem has been detected. The default value is 10 seconds. During this time, TAPI sends initialization strings to reset the modem. If not granted an appropriate time interval, the modem will not reset properly.

For a complete listing of the configuration variables related to the Modem Manager, please refer to "Application Properties for the Modem Manager".

Sequence of Events for Outgoing Calls

The exact sequence of events depends on whether the modem is calling out using data mode or audio mode.

Related Information:

... Data Call

... Audio Call

...Sequence of Events for Incoming Calls

Data Call

1. The originating tag calls the Modem Manager's MakeCall() method.
2. Immediately, the tag's DataPort variable is set to a valid value.
3. Shortly after, DataPort becomes an integer value >=0.

Should the call fail in any way, then DataPort becomes negative. For further details of these values, see Call Progress and Error Codes.

4. If the call setup completes successfully, then DataPort changes to a Stream value (ValueType(DataPort)==8).
5. The call requestor may now read and write to that stream to communicate with the called party.
6. To hang-up the call, you may call CloseStream().
   If the other end hangs-up or the call fails, then DataPort becomes Invalid.

If the call setup fails (Step 3), then the call is retried according to the configured retry settings. If the call is retried, then DataPort becomes an array pointer while the call is queued. DataPort will not go Invalid until the call has been abandoned.

Audio Call

1. The originating tag calls Modem Manager's MakeCall() method with a media value mode indicating an audio call.
2. Immediately, the tag's DataPort variable is set to a valid value.
3. Shortly after, DataPort becomes a pointer to an array, indicating that the call is queued.
4. Once call setup is started, DataPort almost immediately becomes a valid stream value.

This is different from data call setup, in that there is no progress indication. Also, although DataPort is a stream value, the call has not actually connected at this stage. This anomaly arises because a modem operating in voice mode cannot acquire and interpret the various tones that indic-

ate the progress of the call – the modem returns connected status as soon as dialing is complete.
The only practical action the caller can invoke is to operate a time-out that allows a typical connect time, before starting to use the stream.
Any data written to the stream is converted to speech using the voice identified when the discriminator was registered. The speech device is initialized using the string defined by the DialerSpeechInit application property. Data written may include any escape sequences meaningful to the TTS engine.
If a bookmark is set, when the bookmark is reported back to the Modem Manager, a single character equal to 0x45+Bookmark Number is inserted into the stream and may be read by the tag.
The only other data that can be received are DTMF tones input at the remote device. In order to receive such tones, this requirement must have been specified in the media mode supplied to MakeCall(). This data appears in the stream as the digits 0..9, or as the number-hatch (#) or asterisk (*) character.
To shut down a call, you can call the CancelCall() method. If the other end hangs up, DataPort will become invalid.

Allocating Modems in a Managed Pool for Outgoing Calls

There are situations that require a caller to indicate that only modems with a specific property should be used when making a call, even though those modems are part of the central managed pool (e.g. where virtual modems have been set up with special initialization strings (e.g. for cellular connection)). The Area parameter for modem tags can be used for this purpose. If a modem tag has an area defined, then it will only be selected for an outgoing call if the call originator has specified the same area name. Additionally, when allocating local modems on a particular system, the allocator will take account of any Area name associated with

the modem tag, and use it to map the physical modem. The actual capabilities of a modem (Data, Voice, Fax, etc.) are handled by matching the physical media mode of the modem.
See also: Modem Tags.

Local Modems

Local modems are used in circumstances where you wish to be able to make outgoing calls from a modem that is part of the local PC's setup (in a distributed application, for example).
A local modem is defined by creating a modem tag in which the workstation parameter is blank (i.e. leave the "Workstation Name" field of the modem tag's configuration folder blank). The Modem Manager interprets this to mean that it has to associate this logical modem with the first modem that is not otherwise allocated on the local PC. If there is no such modem, then the local modem is unavailable at that workstation.
If you wish to associate a logical modem with a specific local modem, then include a line similar to the following in the workstation.Dynamic file on the PC.

```
ModemTagNameDevice = Actual Modem Device Name
```

Example:

```
Modem01Device = HSP56 Micro Modem
```

Please note the following details:

- A local modem is used for outgoing calls only.
- Local modems are only used on specific instructions from the tag that originates the call (see MakeCall()).

Calls scheduled for handling by a local modem are queued on a separate queue at the local PC. If the local PC is shutdown before the call has been made, then that call will not be picked up by another PC and all memory of that call request will be lost.

Modem Manager Alarm and Event Reporting

The Modem Manager maintains the Value parameter of the Modem Tag to indicate overall status. The value may be one of:

0	Idle
1	Incoming call
2	Outgoing call
3	Error

An alarm may be configured on the Modem tag's value. Configuration of alarm tags on modems is discussed in Modem Tags.

Related Information:

...Internal Event Recording

Internal Event Recording

The Modem Manager records events internally at one of four levels of detail. All events may be displayed on-screen using the Modem Tools widget. The default is to display level 0 events only.

- Level 0 events are selected to be meaningful to normal operations personnel.
- All level 0 events are distributed across all machines for display purposes.
- All level 0 events cause an event entry in the system's alarm/event log.

Level 1..3 events are of a diagnostic nature, and it is not recommended that they be displayed on-screen during normal operation.

- Level 1..3 events are only recorded on the workstations on which they occur.
- All Modem Manager events are recorded in the system's "VTSTrace.txt" file if recording of Modem Manager events is enabled.

The level of on-screen display and the size of the on-screen display history are customizable using application properties. For a complete listing of the application properties related to the Modem Manager, please refer to "Application Properties for the Modem Manager".

Note: You can disable the logging of modem manager alarms with the configuration variable ModemAlarm.

Modem Manager API

If you are writing a custom driver to accept incoming modem calls, then you must use the Modem Manager interface. Fundamental tasks include:

Register a modem

You must call the Modem Manager's Register subroutine, passing the driver's discriminator object value, its station number, and a priority relative to other drivers. Here is a sample call with a priority setting of 10:

```
\ModemManager\Register(Root, Station, 10 { Priority });
```

Unregister a modem

If a driver needs to change its Station address, it must first unregister. You must call the Modem Manager's Unregister subroutine, passing the driver's discriminator object value, and the station number with which it previously registered.

```
\ModemManager\Unregister(Discriminator, Station);
```

Provide a discriminator subroutine

You must provide a discriminator subroutine. The Modem Manager will call this subroutine when it offers you an incoming call. The Modem Manager passes a BUFFER as a parameter. This Buffer contains the initial data received from the line (see HelloPacketLength). You should parse this data and decide whether or not your driver supports it, and for which STATION it is intended.

- Return INVALID to reject this call.
- Return a valid STATION number to accept the call.
- If you accept the call, then the Port\IsConnected() module will go true. You can then obtain the serial port semaphore Port\Sem() to read and write data via the serial port.
- If Port\IsConnected() becomes false, the call has been disconnected.
- If you wish to hang up the call, call the subroutine Port\CallComplete().

Monitor call progress

The Port\DataPort variable is VALID if a call is being setup or is active. At all other times, it is INVALID. During call setup, the value is a SHORT INTEGER indicating the progress of the call, changing to a STREAM when the call connects (this is what IsConnected() indicates).

Related Functions:

... ModemStream

Log events

If required, you may insert entries into the Modem Manager's event log. To do this, call the EventLog subroutine as follows:

```
\ModemManager\EventLog("Text message to be logged");
```

You should set the application property ModemManagerLogSize to at least 256. Failing to do so will result in the loss of all logging events that occur prior to the first display of the event log.

Related Information:

...Required Subroutines in Custom Drivers

...Modem Manager Functions

... ModemControl Plug-in

...Call Progress and Error Codes

... Modem Manager Constants

... Modem Manager Properties

...Example Audio Discriminator

...Example Data Discriminator

Required Subroutines in Custom Drivers

The following topics describe the Modem manager subroutines that must be used if you are writing a custom driver to accept incoming modem calls.

Registering a modem

You must call the Modem Manager's Register subroutine, passing the driver's discriminator object value, its station number, and a priority relative to other drivers. Here is a sample call with a priority setting of 10:

```
\ModemManager\Register(Root, Station, 10 { Priority });
```

Discriminator

You must provide a discriminator subroutine. The Modem Manager will call this subroutine when it offers you an incoming call. The Modem Manager passes a BUFFER as a parameter. This Buffer contains the initial data received from the line (see HelloPacketLength). You should parse this data and decide whether or not your driver supports it, and for which STATION it is intended.

- Return INVALID to reject this call.
- Return a valid STATION number to accept the call.
- If you accept the call, then the Port\IsConnected() module will go true. You can then obtain the serial port semaphore Port\Sem() to read and write data via the serial port.
- If Port\IsConnected() becomes false, the call has been disconnected.
- If you wish to hang up the call, call the subroutine Port\CallComplete().

Call Progress

The Port\DataPort variable is VALID if a call is being setup or is active. At all other times, it is INVALID. During call setup, the value is a SHORT INTEGER indicating the progress of the call, changing to a STREAM when the call connects (this is what IsConnected() indicates). For details on the integer values, please refer to the ModemStream script function.

Event Log

If required, you may insert entries into the Modem Manager's event log. To do this, call the EventLog subroutine as follows:

```
\ModemManager\EventLog("Text message to be logged");
```

Note: You should set the application property ModemManagerLogSize to at least 256. Failing to do so will result in the loss of all logging events that occur prior to the first display of the event log.

Unregister (Modem Manager)

If a driver needs to change its Station address, it must first unregister. You must call the Modem Manager's Unregister subroutine, passing the driver's discriminator object value, and the station number with which it previously registered.

```
\ModemManager\Unregister(Discriminator, Station);
```

Modem Manager Functions

The following functions are related to the Modem Manager.

CallerID	Returns the callerID string from the TAPI LINECALLINFO structure. *Works only with the Unimodem driver, not with the Trihedral TSP driver*.
CancelCall	Removes a queued call or abandons a call that is in-progress
Fail	Tells the Modem Manager to abort and retry an established, outgoing call
FindModem	Returns a pointer to one of the Modem Manager's own internal modem objects. This pointer may then be used to access public, read-only properties for display purposes
MakeCall	Queues a call request
ModemCount	Number of available modems
ModemDev	Obtain wave device handle
ModemDial	Initiate an outgoing call
ModemDigits	Monitor a voice call for key presses
ModemList	Enumerate the available modems and their capabilities
ModemMedia	Determine or change the media mode of a call
ModemStream	Prepare to receive an incoming call

ModemTransfer	Transfer a call to another system service
Register (Modem Manager)	Registers a discriminator that accepts incoming calls

Note also, that the Connect() method of the Serial Port tag includes the parameter, ConnectInitString, which can be used to programmatically send an initialization string for any particular connection attempt. If valid, this will override the initialization string configured in the serial port. A "\r" is always appended to this string.

ModemControl Plug-in

The \ModemControl plug-in module expects four parameters as follows:

Function

Identifies the calling condition for the plug-in:

Calling Condition	Definition
0	about to begin a dial attempt for an outgoing call
1	completion of a successful outgoing call
2	failure of an outgoing call
3	modem is going idle after a call or on exit from a failed mode
4	modem is being set into a failed mode

Index

The index of this modem device in the list of system modem devices (as returned by the ModemList function).

Tag

The name of the tag that originated the call (functions 0, 1, and 2 only).

UserData

The opaque user data supplied as a parameter to the MakeCall() method (functions 0, 1, and 2 only).

Call Progress and Error Codes

Values greater than or equal to zero are progress codes, while values less than zero are errors.

Progress Codes

Code	Description
0	LineIdle Idle (no connection); waiting for call
1	CallBegins Starting outbound call
2	RingAnswered Incoming call answered; no connection yet
3	DialToneOK Dial tone on outbound call
4	DialingOut Dialing outbound call
5	RemoteRinging Remote phone ringing on outbound call
6	RemoteBusy Remote phone busy on outbound call
7	CallConnected Connected (you should not see this value; stream value is returned at this point)
8	HandedOff Another application is handling the call
9	IncomingCallRQ Incoming call detected
10	TempProblem Temporary modem problem – hang up and retry
11	IncomingCall Incoming call accepted

Error Values

Value	Error
-1	NoModems No modems defined in the system
-2	NoDialTone No dial tone detected on outbound call

-3	RemoteBusyErr Remote phone busy on outbound call
-4	NoAnswer No answer on outbound call
-5	CallerHungUp Other end hung up an incoming call
-6	ModemUnavail Modem unavailable
-7	CallFailure Other call termination condition

TAPI Errors

Errors equal to or less than -101 correspond to the Microsoft Telephony Interface (TAPI) errors, which are normally numbered from 1. Specific errors that may be encountered are:

Code	TAPI error
-101	TAPI error code 1 (LINEERR_ALLOCATED) – the serial port is in exclusive use by some other process.
-112	TAPI error code 12 (LINEERR_INCOMPATIBLEAPIVERSION) – the system does not have the required version of telephony support (TAPI 2.0 required).
-115	TAPI error code 15 (LINEERR_INUSE) – the line device is in use and cannot currently be configured, nor can it allow a party to be added or a call to be answered, placed, or transferred.
-147	TAPI error code 47 (LINEERR_INVALMEDIAMODE) – the requested media mode could not be accommodated (e.g. voice call request on a non-voice modem).
-167	TAPI error code 67 (LINEERR_NODEVICE) – the specified device identifier, which was previously valid, is no longer accepted because the associated device has been removed from the system since TAPI was last initialized. Alternately, the line device has no associated device for the given device class.
-168	TAPI error code 68 (LINEERR_NODRIVER) – either TAPIAddr.dll could not be located, or the telephone service provider for the specified device found that one of its components is missing or corrupt in a way that was not detected at initialization time. Use the Telephony Control Panel to correct the problem.
-169	TAPI error code 69 (LINEERR_NOMEM) – insufficient memory to perform the operation, or unable to lock memory.
-175	TAPI error code 75 (LINEERR_RESOURCEUNAVAIL) – insufficient resources to complete the operation (e.g. a line cannot be opened due to a dynamic resource overcommitment). Also occurs where a modem is being used by a non-TAPI applic-

	ation.

The full list of possible TAPI error codes:

Error	Code	Error	Code
ALLOCATED	101	INVALMEDIAMODE	147
BADDEVICEID	102	INVALMESSAGEID	148
BEARERMODEUNAVAIL	103	INVALPARAM	150
CALLUNAVAIL	105	INVALPARKID	151
COMPLETIONOVERRUN	106	INVALPARKMODE	152
CONFERENCEFULL	107	INVALPOINTER	153
DIALBILLING	108	INVALPRIVSELECT	154
DIALDIALTONE	109	INVALRATE	155
DIALPROMPT	110	INVALREQUESTMODE	156
DIALQUIET	111	INVALTERMINALID	157
INCOMPATIBLEAPIVERSION	112	INVALTERMINALMODE	158
INCOMPATIBLEEXTVERSION	113	INVALTIMEOUT	159
INIFILECORRUPT	114	INVALTONE	160
INUSE	115	INVALTONELIST	161
INVALADDRESS	116	INVALTONEMODE	162
INVALADDRESSID	117	INVALTRANSFERMODE	163
INVALADDRESSMODE	118	LINEMAPPERFAILED	164
INVALADDRESSSTATE	119	NOCONFERENCE	165
INVALAPPHANDLE	120	NODEVICE	166
INVALAPPNAME	121	NODRIVER	167
INVALBEARERMODE	122	NOMEM	168
INVALCALLCOMPLMODE	123	NOREQUEST	169
INVALCALLHANDLE	124	NOTOWNER	170
INVALCALLPARAMS	125	NOTREGISTERED	171
INVALCALLPRIVILEGE	126	OPERATIONFAILED	172

INVALCALLSELECT	127	OPERATIONUNAVAIL	173
INVALCALLSTATE	128	RATEUNAVAIL	174
INVALCALLSTATELIST	129	RESOURCEUNAVAIL	175
INVALCARD	130	REQUESTOVERRUN	176
INVALCOMPLETIONID	131	STRUCTURETOOSMALL	177
INVALCONFCALLHANDLE	132	TARGETNOTFOUND	178
INVALCONSULTCALLHANDLE	133	TARGETSELF	179
INVALCOUNTRYCODE	134	UNINITIALIZED	180
INVALDEVICECLASS	135	USERUSERINFOTOOBIG	181
INVALDEVICEHANDLE	136	REINIT	182
INVALDIALPARAMS	137	ADDRESSBLOCKED	183
INVALDIGITLIST	138	BILLINGREJECTED	184
INVALDIGITMODE	139	INVALFEATURE	185
INVALDIGITS	140	NOMULTIPLEINSTANCE	186
INVALEXTVERSION	141	INVALAGENTID	187
INVALGROUPID	142	INVALAGENTGROUP	188
INVALLINEHANDLE	143	INVALPASSWORD	189
INVALLINESTATE	144	INVALAGENTSTATE	190
INVALLOCATION	145	INVALAGENTACTIVITY	191
INVALMEDIALIST	146	DIALVOICEDETECT	192
ALLOCATED	101	INVALMEDIAMODE	147

Modem Tag Return Values

The modem tag type associates a specific modem on a workstation with a physical phone line. A modem tag can have eight possible return values:

Modem Tag Value	Description
0	Idle
1	Modem calling (outgoing call in progress)
2	Modem answering (incoming call in progress)

3	General failure (modem failed)
4	Server unavailable
5	Workstation owning the modem is unavailable
6	External failure
7	Manually disabled

The value of the modem tag can be used with other tags; for example a modem tag could be selected as the tag to be monitored by an alarm tag. An alarm could be triggered if the value of the modem tag is 3 (modem failed), alerting the users to modem failure.
See: Modem Tags.

Modem Manager Constants

The following constants are associated with the media handling properties and requirements of modems, call requests, and incoming discriminators.

AUDIO 0x08 for use with speech synthesis
DATAMODEM 0x10 for data transfer

Three additional constants are defined, although the Modem Manager has no specific support for them.

UNKNOWN 0x02
VOICE 0x04
G3FAX 0x20

Additional control properties associated with call handling are:

NEED_DTMF 0x1000 enable receipt of DTMF tones when handling an audio call
DTMF_STOPS_SPEECH 0x2000 received DTMF tones interrupt speech

Modem Manager Properties

The following variables may be accessed in read-only mode:

Variable	Description
AvailLinesNumber	Phone lines still operational

CallAttempts	Total call attempts
CallCount	Number of calls placed
CallDelay	Total accumulated delay between call requested time, and actual time handled
ConfigChange	Trigger. Set whenever the modem configuration changes
FailCount	Number of calls that have failed
LocalCallAttempts	Total local modem call attempts
LocalCallCount	Number of local modem calls placed
LocalCallDelay	Total accumulated delay between call requested time and actual time handled (local modems)
LocalFailCount	Number of local modem calls that failed
UsedLines	Number of phone lines in use

Example Audio Discriminator

This example is of source code for a VTScada page that implements a simple voice response system.

```
[
  Title = "Voice";
  Color = "<E6E6E6>";
  Bitmap;
  SecBit;
  WinFlag = 0;
  AudioDiscriminator Module { Module to receive incoming calls };
  { TAPI Media modes used to select the compatibility of a modem }
  Constant TAPIMEDIATYPE_AUDIO = 0x08;
  Constant TAPIMEDIATYPE_DATAMODEM = 0x10;
  Constant NEED_DTMF = 0x1000 { Enable DTMF when handling call };
  Constant DTMF_STOPS_SPEECH = 0x2000 { Rec'd DTMF stops any speech
};
  spVoice = "bf5ead45-9f65-11cf-8fc8-0020af14f271"
  { Voice for telephone modem};
  Constant SMALLPAUSE = "\Pau=250\";
  Constant LONGPAUSE = "\Pau=500\";
  Trig;
  Number;
  Name = "(Phone)";
  DataPort;
  KeyPress;
  BookMark;
```

```
  Phase;
  Phrase;
  Incoming = 0;
]
Init [
   If \ModemManager\Started Main;
   [
      Return(Self);
      AddVariable(\Code, Name, 0, 256, 0, 0, 0, 0, 0, 0);
      Scope(\Code, Name) = Self();
      Phase = 0;
      \ModemManager\Register(Self(), 1, 1,
      TAPIMEDIATYPE_AUDIO + NEED_DTMF + DTMF_STOPS_SPEECH,
      SpVoice, 30 { Discriminator Timeout }, Name);
   ]
]
Main [
   Return(Self);
   \System\Edit(40, 60, 180, 90, "Number to Call", Number, !Valid
(DataPort), Trig, Invalid, 4, 1);
   If Trig;
   [
      Trig = Invalid;
      Phase = BookMark = 1;
      CloseStream(DataPort);
      \Code\ModemManager\MakeCall(Number, 110, 8, 1, 0, 5, Name,
Invalid,
      TAPIMEDIATYPE_AUDIO + NEED_DTMF + DTMF_STOPS_SPEECH, spVoice);
   ]
   If TimeOut((ValueType(DataPort) == 8) && !Incoming, 6) && Phase ==
BookMark;
   [
      BookMark = -1;
      IfThen(!Valid(Case(Phase - 1,
                  {0} Phrase = "Hello and welcome. Please press a
key to continue.",
                  {1} Phrase = "This is message number 1",
                  {2} Phrase = "This is message number 2",
                  {3} Phrase = "This is message number 3",
                  {4} Phrase = "Goodbye!"
                  )),
                  CloseStream(DataPort)
            );
      SWrite(DataPort, "%s%s\\Mrk=%d\\\r", SmallPause, Phrase,
Phase);
   ]
   If SerWait(DataPort, 1);
   [
      KeyPress = SerRcv(DataPort, 1);
      IfElse( PickValid(KeyPress >= 0 && KeyPress <= 9, 0) ||
KeyPress == "#" || KeyPress == "*", Execute(
      { DTMF from phone - cancels any pending speech }
      Phase++;
      IfThen(KeyPress == "*",
            Phase = 1;
      );
      IfThen(KeyPress == "#",
```

```
            Phase = 5;
      );
      BookMark = Phase;
   );
{ Else }
{ Must be a bookmark }
   IfThen(KeyPress == MakeBuff(1, 0x41 + Phase),
      IfThen(Phase > 4,
               Phase++;
            );
           BookMark = Phase;
      );
   );
]
]
<
{=================== Voice\AudioDiscriminator ================== }
{ This module, runs the receive call handling. If a call is not suc-
cessfully authenticated then the caller
ModemManager) will time out and will then attempt to hand the call
off. }
{=================================================================-
=}
AudioDiscriminator
(
   Mode;
)
Check [
   Incoming = 1;
   If TimeOut(1, 7);
  [
    Phase = 1;
    Phrase = "Hello, you have reached the VTScada modem tester.";
    SWrite(DataPort, "%s%s\\Mrk=%d\\\r", SmallPause, Phrase, Phase);
  ]
  If KeyPress == "9" Idle;
  [
    Phase = BookMark = 1;
    Incoming = 0;
    Return(1);
  ]
]
Idle [
]
{ End of Voice\AudioDiscriminator }
>
```

Example Data Discriminator

The following example displays a minimalist discriminator for the standard Modbus Compatible Device. This code would typically be in a file in the application directory, and referenced in the SERVICES section of the application's AppRoot.src root file.

```
[
Discriminator Module;
Connect Module;
Root;
Name;
Port;
]
Init [    { This module registers as the discriminator for tag MB01 }
   If \ModemManager\Started Main;
   [
   Name = "MB01";
   Port = Scope(\Code, Name)\Port;
   Root = Self;
   \ModemManager\Register(Root, Name, 10);
]
]
Main [
]
<
{================ ModiconDriver\Discriminator ====================}
{ This module pass in a buffer, return a Station ID if accept the }
{ call. The station address in the incoming message is inspected. }
{ Return invalid if reject the call.                               }
{================================================================}
Discriminator
(
Buffer;
)
Main [
   If watch(1);
   [
      IfElse(PickValid(GetByte(Buffer, 0) == Scope(\Code, Name)\Sta-
tion, 0),
      Return(Name),
      { Else }
      Return(Invalid)
      );
   ]
]
{ End of ModiconDriver\Discriminator }
>
<
{===================== ModiconDriver\Connect ====================== }
{ Returns the Workstation name of the driver server. Called by the  }
{ Modem Manager to determine the machine name of the current server }
{ for this driver.                                                  }
{================================================================ }
Connect
(
Station;
Stream;
)
Main [
   If watch(1);
   [
      Return(WkStaInfo(0)) ;
   ]
```

```
]
{ End of ModiconDriver\Connect }
>
```

TAPI and UniModem Considerations

The Modem Manager does not interface directly with modems; instead, it uses the system-provided telephony interface, TAPI. This enables VTScada to share modems with other TAPI-compliant applications, such as RAS services, FAX services, and so forth.
All TAPI modems share the same driver called UniModem. When a modem manufacturer's driver is installed, what actually happens is that an .inf file is processed, and a number of registry entries, understandable by UniModem, are added.

Modem Initialization Strings

The Modem Manager has no knowledge of modem initialization strings. These come from the system registry where they are entered when the modem manufacturer's driver is installed. The Windows Control Panel's Phone and Modems option (or Modems option) provides a mechanism for adding extra initialization strings.

Baud Rate

At first sight, it would appear from the Modem Manager's MakeCall method that the baud rate and other communications parameters can be set in the parameters to this method. However, what is being set is actually the parameters pertaining to local communication with the modem, not parameters for communication between the local and remote modem.
The only way that particular parameters for the modem – modem link can be set is by setting a specific initialization string for the modem.
If you want to use specific parameters for different destination locations, then it is necessary to define cloned "ghost" modems in the system configuration. This is discussed in the section that follows.

Cloned Modems

It is possible to define multiple modem entries that refer to the same physical modem. Different initialization strings can then be associated with the different clones. Note that if using this method, you probably have to reboot the system after adding each modem before you can add another.

In order to use these cloned modems from VTScada, you have to be able to specify which modem (or type of modem) to use for a particular call. This is achieved using the Area parameter to the modem tag, and the MakeCall method. Normally, the area parameter to a modem tag is unused. If it is used, then the Modem Manager will only select that modem for an outgoing call if the area matches the area parameter supplied to MakeCall.

Note that since a set of cloned modems actually map to one physical modem, only one of them can be in use at any time.

Further, only one of the clones may have incoming calls enabled. Failure to observe this will result in an error opening any of the modems in the clone set.

TAPI Errors

If you find that VTScada reports a modem as permanently unavailable, you may find that the modem event log indicates that a TAPI error is occurring. While a full list of such errors is given in Call Progress and Error Codes, in the vast majority of cases, this will be caused by one of two things:

- The system has not been rebooted since the modems were last reconfigured. While a reboot is not always required, after some reconfiguration actions the system cannot enumerate the resources correctly until a reboot has occurred.
- Some non-TAPI-compliant application has opened the serial port associated with the modem. Note that this includes VTScada's ComPort function and the system's HyperTerminal application (commonly used in troubleshooting modems).

RPC Manager Service

This chapter provides architectural, programming, and configuration information on the VTScada Remote Procedure Call (RPC) subsystem. This section assumes good knowledge of the VTScada scripting language, programming, and familiarity with networking concepts sub-sections:
The following terms and abbreviations are used when referring to the RPC manager and its functions.

FIFO	The FIFO acronym stands for "First-in, First-out"; a simple queue.
RPC	Remote Procedure Call. A Remote Procedure Call is simply an invocation of a VTScada module (subroutine or launched) on one workstation from VTScada code running on the same workstation or on another workstation.
	Note: It should be noted that the "RPC" acronym does not pertain to the operating system's remote procedure call mechanism.
RPC Manager	The generic name given to the components comprising the RPC subsystem within VTScada.
Service	An object within a networked VTScada system that has an instance on other workstations on the network.
Service Instance	An instance of a Service. There can be one of these per application per workstation on the network.
Service Synchronization	The ability to ensure that all Service Instances of the same Service are operating on identical copies of the Service's data.
Synchronizable State	The data on which a Service operates, and which must be identically replicated in each instance of that Service within the distributed domain.

Related Information:

...Overview of the RPC Manager Service - RPC Manager enables VTScada to operate within a domain distributed across multiple machines, by providing the ability to remotely execute a VTScada subroutine.

...Services - RPC Manager provides the mechanism for ensuring that services remain synchronized across the distributed domain.

...Connection Configuration and Management - how RPC Manager maintains the inter-machine connections in a distributed system.

...Configure Cross-Application RPC - how cross-application RPC works and the modifications you will need to make to your application to achieve this.

...Application Control of Servership - in rare cases, it may be necessary to control the servership of one or more services by the application itself.

...System Level Services - system level services can be used to provide a service that is common to many applications or provide a service that does not rely on the presence of an application.

...API Reference - a reference for each method that RPC Manager provides for external use.

...Diagnostics - See: Trace Viewer Application.

...RPC Routing and Execution - internal and external.

...RPC Security - RPC security is system based and is concerned with ensuring that RPC communication between VTScada servers is secure.

...Configuration - two types of initialization data used by RPC Manager.

...Protocol - TCP and alternate communications protocols used by RPC.

Overview of the RPC Manager Service

RPC Manager enables VTScada to operate within a domain distributed across multiple machines, by providing the ability to remotely execute a VTScada subroutine.

The RPC Manager provides the cornerstone of a comprehensive server-/client architecture, which enables tags and other VTScada services running on physically separate workstations to coordinate their activities and share data. VTScada contains a number of services, (such as the Alarm Manager, Log Manager, Modem Manager), and enables these services, all of the I/O drivers and other network-aware software components to share their data using the RPC Manager.

A VTScada service is simply a VTScada module that registers itself as a named component with the RPC Manager. A VTScada service encapsulates the data and processing logic that provides a specific function within the application. Each service should be regarded as a separate entity whose service name is unique within the application. Different applications running on the same workstation can contain a service of the same name as a service in another application, That is, two standard VTScada applications will, for example, each contain a service called AlarmManager.

A service running within an application instance is termed a service instance. A distributed system contains many instances of the same service, each one running on a separate physical machine. At any given time, for each service, there is exactly one service instance that has "server" status. All other instances of the same service (running on machines other than the server instance) are either clients to that server or have not yet determined what their status should be. A brief moment may exist when there is no server for a service, but there will never be more than one server.

Server status can shift from one service instance to another at any time. Such shifts are normally the result of some failure of the server or failure of the server to communicate with an external device. When such shifts occur, each service must arbitrate which service instance is going to assume server status. All other service instances must then resynchronize to the new server and so become clients of it.

The RPC Manager provides the code that performs service synchronization and manages servers and service instances. The service

code need only provide a few, simple methods that will be invoked by RPC Manager to achieve this. A simple, yet functional service example that supports all the essential RPC Manager methods is discussed in "Create a Simple Service".

RPC Manager also provides a set of methods that provide services with the ability to make RPCs, as well as a number of useful, ancillary methods. All the methods available are documented in the Application Control of Servership

In normal operation, RPCManager controls which service instance is currently the server for a service. The system is simply configured to instruct RPCManager how its servership control algorithms should operate.

In rare cases, it may be necessary to control the servership of one or more services by the application itself. Consider, for example, a system where maintenance of the system is necessary while the application is running. For operational reasons, the system owners do not want a number of key services to have servership held by the machine undergoing maintenance, but do want the application to be running.

In such circumstances the application needs to either be able to disable a particular machine from owning servership of the key services or, more generally, be able to control which machine is the server for those services. More complex situations could arise where, for example, a specific machine does not want to be considered a candidate for servership.

References in this document to version 4 of RPCManager refer to the RPCManager capable of using version 4 RPCManager protocol. This applies to VTS version 10 onwards.

Related Information:

...RPC High Level Design - diagram showing RPC module organization.

... Remote Procedure Calls (RPCs) - notes on how calls are threaded.

...Session IDs - definition of and uses for.

...Types of RPC - directed RPC versus service RPC.

...Cross-Application RPC - notes

...Permitted Data Types in RPC - reference: list of types that may be used in RPC parameters.

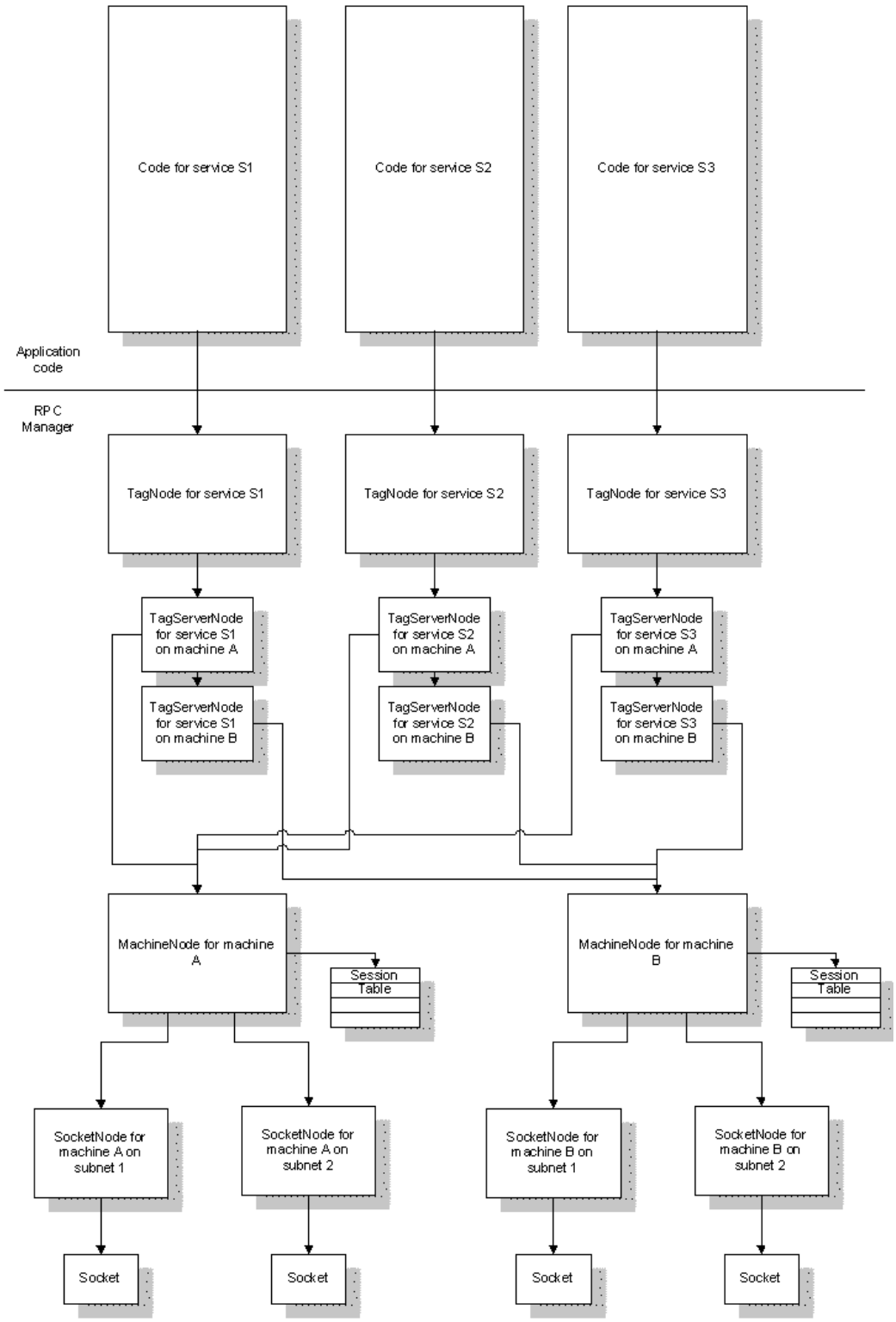
...Compression - notes

...Packed RPC Streams - to ensure that calls are made in a given sequence.

...Application Settings for RPC - options for control.

RPC High Level Design

RPC Manager is composed of a number of layers, each consisting of a VTScada object instance. An illustration of the layers of which the RPC Manager follows:



Machines participating in the distributed domain communicate using sockets over TCP/IP. Each machine can be identified by any of the names by which it is known, or any of the IPs by which it can be addressed. RPC

Manager on each machine runs a socket server which listens for socket connection requests on a specific port. The port number is configurable and defaults to 5780.
The RPC Manager layer "nearest" the sockets consists of SocketNode object instances. There is one SocketNode instance for each IP address with which the local RPC Manager is communicating. The next layer up consists of a set of MachineNode object instances. There is one MachineNode object instance for each machine participating in the distributed domain. Where all machines within the domain have only one IP address, there will be a one-to-one correspondence between MachineNodes and SocketNodes. However, where a machine has more than one IP address (otherwise known as a multi-homed system), there will be more than one SocketNode for the machine's MachineNode. SocketNode provides a reliable error-free transport for RPCs via a specific IP address. MachineNode provides the buffering and routing of RPCs to its sub-ordinate SocketNodes.
On a single-homed system, the RPC transport will fail when a MachineNode's single sub-ordinate SocketNode is unable to reliably transport RPCs to the intended destination. In a multi-homed system, the RPC transport will fail only when all sub-ordinate SocketNodes are unable to reliably transport RPCs.
To identify the integrity of an RPC transport to higher layers, MachineNode maintains a session identifier (SID) (see Session IDs) for each application instance that is running on the remote machine associated with that MachineNode.
The SID is opaque data. You should not try to interpret the contents of it, just accept that the SID is unique to the reliable RPC transport to/from the remote application instance. If the transport fails, the SID will change to a new, unique value. If the remote application instance is terminated and a new one started, the SID will change to a new, unique value. If no RPC transport can be obtained to a specified remote application instance, (either because the application is not running or communication has failed), the SID will have a unique value, defined by RPC Manager as RPC_

NO_SID. That the SID remains at an unchanging, valid value is an absolute guarantee that the same remote application instance is being communicated with and that no loss of RPCs is occurring.
The next higher layer, which manages service instances, uses the SID to control service synchronization. A TagNode object instance exists for each service instance that is running on the local machine. Each TagNode has a number of TagServerNode object instances, one for each machine that can be a server for the service (this may include the local machine). The TagServerNode remembers the SID that it obtains when it establishes contact with the corresponding machine and uses any change in the SID to detect loss of reliable communication with the potential server. Any change in SID, for the TagServerNode that corresponds to the server instance, will require the TagNode to initiate service re-synchronization. In this way, any loss of the information flowing via RPCs from a server to a client is detected and the service automatically re-synchronized to ensure that the client is not "out-of-step" with the server. This process is termed "service synchronization".
VTScada contains a variable, located in the root (system) scope that can be used as a prefix to access any of the methods and public data that RPC Manager provides. This variable is \RPCManager. You will see this prefix used repeatedly throughout the remainder of this document.

Remote Procedure Calls (RPCs)

When a VTScada subroutine executes, all other statement execution within the thread that is running the subroutine is suspended. The RPC subroutine is run on the RPC Manager's thread. This is a deliberate behavior, ensuring that only one RPC subroutine is ever running within an instance of VTScada. This enables the RPC subroutine to access three important variables, which RPC Manager maintains:

- CurSocketNode – the MachineNode object that exists for the machine that issued the RPC. A MachineNode exists for every machine that an instance of VTScada can communicate with, including the local machine. Note that, before support for multi-homed machines was introduced, this variable did

indeed refer to the SocketNode from which an RPC request had been received. Its name has not been changed, to preserve compatibility with existing applications.

- CurSessionID – the opaque Session ID of the application instance that issued the RPC .
- CurSourceAppGUID – the GUID of the application that issued the RPC .

Note: The contents of these three variables are valid only for the duration of the execution of the RPC subroutine. When the RPC subroutine returns, these variables are deliberately invalidated and then set to new contents for the next RPC subroutine execution. If you wish to retain the contents of either of these variables, you must make a copy during execution of the RPC subroutine.

If the RPC subroutine is not written as a launched module rather than as a subroutine, then RPC Manager will launch the module, but access to the above variables is not permitted. A module launched in this way will be launched on the thread of the context into which the module is launched (normally the service's thread).

You should treat RPC subroutines in the same manner as one would treat interrupt service routines. Execution of an RPC subroutine suspends all other RPC activity on the machine executing the subroutine. Therefore, if there is time-consuming work to be performed, it is preferable to have the RPC subroutine launch a worker module into an appropriate context, passing the contents of CurSocketNode and CurSessionID if required, as parameters. For example:

```
{=================== SampleRPCLaunch ======================}
{ This subroutine is called via RPC and launches off a worker }
{ module to do some time-consuming work.                    }
{=========================================================}
(
  SomeParam           { Parameter to the RPC subroutine
};
)
[
  DoTheWork Module;
]

Only [
  If 1;
  [
```

```
    {** Pass the CurSessionID & CurSocketNode to the launched module
**}
    DoTheWork(SomeParam, \RPCManager\CurSocketNode, \RPCMan-
ager\CurSessionID);
    Return(0);
  ]
]

<
{============== SampleRPCLaunch\DoTheWork ==================}
{============================================================}
DoTheWork
(
  SomeParam             { Same as RPC subroutine
};
  ClientSocketNode      { Client socket node at time of starting
};
  ClientSID             { Calling client's session ID
};
)

Init [
  If 1 Main;
  [
    { Typical initialization code here }
  ]
]

Main [
  { Whatever code performs the time consuming work }
  If Finished;
  [
    Slay(Self(), 0);  { Terminate myself }
  ]
]

{ End of SampleRPCLaunch\DoTheWork }
>
```

Related Information:

Cross-Application RPC

Session IDs

RPC Manager maintains a Session ID (SID) for each instance of an application running on a remote machine. This SID should be treated as "opaque" data. It can be tested for equality or inequality with another SID, it can be tested against Invalid and it can be tested for equality or inequality against the constant \RPCManager\RPC_NO_SID.

RPC Manager provides a SID so that your application can ensure integrity with a remote application instance. A SID will change value whenever

reliable communication, for RPC, can no longer be maintained with a remote application instance. This can be caused by:

- Irrecoverable failure of the communication link to the remote machine. RPC Manager will do its best to recover communication, in the event of transient failure, but there will come a point when RPC Manager determines that RPCs can no longer be reliably delivered to the destination. In this case, the current SID will change to RPC_NO_SID.
- Termination of the application instance on the remote machine. SIDs will indicate the termination and restart of a remote application instance that occurs during a transient network failure. In this case the SID will change to RPC_NO_SID, if no application instance is running, or to a new SID if a new application instance is running.

A SID remaining at the same value guarantees that all RPC requests are being reliably transported to the same target application instance.
A SID can be obtained from a number of sources:

- \RPCManager\GetSessionID(). This RPC Manager provided module can be called as a subroutine or run in steady state. It takes an application identifier and a machine identifier as parameters and returns the current SID for that application.
- \RPCManager\GetServerSIDPtr(). This RPC Manager provided module can also be called as either a subroutine or run in steady state. It takes a service name and returns a reference to a variable holding the SID for the application instance that houses the current server for that service.
- \RPCManager\Send(). This RPC Manager provided subroutine is used to issue an RPC. It will return the SID for the remote application instance for directed RPCs, , that was present at the moment the RPC was queued for transmission to the remote machine.
- \RPCManager\CurrentSessionID. This public variable is only valid during the execution of an RPC subroutine and contains the SID for the application instance that sourced the RPC request.

An application can make use of the SID to ensure that it is still communicating with the same instance of an application as it was at some prior time. RPC Manager uses the SID for just such a purpose when synchronizing services .

Related Information:
Types of RPC
Services

Types of RPC

The method used to specify the destination for an RPC can take one of two forms:

- A directed RPC. This type of RPC specifies that the RPC should be executed on a specific machine, identified by name or IP.
- A service RPC. This type of RPC specifies that the RPC should be executed on one or more machines that are participating in an application service (see Services), identifying the machines using terms such as "on the server" or "on all clients".

The service RPC is by far the most common. Performing RPC via services is much easier as RPC Manager does all the hard work of tracking which machine is the server and, most importantly, of maintaining synchronization between machines.

Cross-Application RPC

In most applications, RPCs are made between two instances of the same application. By definition, these will be running on different machines. Some system designs, however, are composed of multiple applications and require communication between the different applications. RPC Manager provides the ability to communicate between applications and supports cross-application services and service synchronization, relieving the system developer of the control code necessary to achieve this.

Permitted Data Types in RPC

The following data types can be used in parameters to RPCs:

- Numerics
- Text
- Streams

- Arrays of the above, up to, and including, a maximum of 3 dimensions.
- Dictionaries and Structures.

Note that pointers, including arrays of pointers are not supported.

Compression

The parameters to an RPC are automatically compressed to reduce the bandwidth requirements for RPC; however, it is good programming practice to pass large quantities of data as infrequently as possible.

Packed RPC Streams

Each RPC is asynchronous. The delivery order of RPCs is guaranteed to be the same as the order in which they were issued from an individual machine.
There means that there is no guarantee that if machine A issues an RPC to machine C moments before machine B issues an RPC to machine C, that the two RPCs will arrive at C in the order A, then B. However, if machine A issues RPC1 and then RPC2 to machine C, machine C will receive and process RPC1 before RPC2.
Where there is a need to ensure that a sequence of RPCs issued by machine A is either executed in its entirety by machine B, regardless of communication interruptions, or that there is a need to ensure that the sequence of RPCs cannot be interposed by RPCs from another machine, the sequence of RPCs can be "packed" into a stream. This is done using the RPC Manager method "PackRPC".
The stream, so obtained is then used as a parameter to a remote "Run-RPC" RPC, a method also provided by RPC Manager.
This facility is commonly used to store a sequence of RPCs that will create a copy of a service's synchronizable state on a remote machine.

Services

RPC Manager provides the mechanism for ensuring that services remain synchronized across the distributed domain. Each machine participating in the domain runs a copy of the service (a "service instance"). Exactly one service instance has server status and is sometimes referred to as the primary server for the service. All others have client status. Any number of machines can be capable of becoming the server. This list of potential servers is specified in the application's Servers.RPC file or via the Edit Server Lists panel in the Application Configuration dialog.

For the purposes of synchronization, a service is assumed to contain a set of data that represents the "synchronizable state" of the service. When the service starts up on another machine and wishes to become a client of the server, the synchronizable state is transferred to the client. The service instance in server mode will then issue updates to that set of data. The client receives these updates and processes them to ensure that the service on the client reflects the state of the service on the server.

Likewise, if server status is to be transferred from one machine to another, each client of the new server must synchronize to the new server.

As the reader can probably appreciate, synchronization is a complex process, with many variants, which must be proof against many possible faults, both logic and timing. To have each service provide its own code to achieve this would be a rich source of application faults. Hence, RPC Manager provides all the necessary control logic to achieve service synchronization and requires each service to only provide a set of operations to supply the synchronizable state, sufficient for the service's needs.

Related Information:

...Programming Example: Create a Simple Service - example code with following explanations.

...Read and Write Locks - when to use semaphores.

...Synchronization Sequence - summary of the topics in this section.

...RPC Call-Backs - using RPCServerNotice to watch for events.

Programming Example: Create a Simple Service

To better understand the development of an RPC service, this section contains the VTScada code for a fully functional, yet simple service, which will maintain its synchronizable state correctly under all circumstances. We will build this code up from scratch, starting with the simplest specification. The code contained within here is complete and fully functional. You can cut and paste the code into source files and use it.
The function of the service is to continually increment one numeric variable, on the server and broadcast the value of that variable to all clients. Each client that receives the new number verifies that it is one greater than the previous one. If it is not, the client increments an error counter. If the service functions correctly, each client will receive the value of the counter as the synchronizable state, followed by incremental updates to the count, regardless of how many clients there are or at what point in the server's life they are started.
The code for the main module of the service is quite straightforward. First, we need to register our service with RPC Manager and then enter a state where, if we are the server the counter is incremented and transmitted to all clients, via an RPC.
Here is the main module, in its entirety, along with an RPC subroutine that it calls:

```
{============================== SampleService
================================}
{ This service maintains a simple synchronizable state and properly
supports  }
{ all entry points required by RPC Manager for correct operation.
}
{=================================================================-
=========}
[
  SvcName = "Sample"            { Deliberately different from the mod-
ule name.
                                  Doesn't have to be different, but
```

```
this allows
                                  clear distinction in the code
between the two};
  RPCStatus                     { Current RPC connect status
};

  CurrentServer                 { Current server
};
  ServiceMode                   { Textual description of service mode
};

  {***** The synchronizable state of this service is represented by a
simple,
         continually incrementing  numeric. The last value generated
by this
         service instance and the last value received by this service
instance
         are needed to ensure continuity of count. *****}
  HeartbeatCountIn = 0          { Heartbeat counter
};
  HeartbeatCountOut = 1         { Heartbeat counter
};

  {***** Internal variables *****}
  LastHeartbeatCount = 0        { Last Heartbeat count rxd
};
  SequenceErrors = 0            { Number of periodic event sequence
errors    };
  ServerEnable                  { Server enable flag...prevents race
condition};

  {***** Regular service RPC modules which are service specific
*****}
  PeriodicRPC Module            { Periodic event RPC call module
};

  {***** Required Synchronization modules, which RPC Manager will
call *****}
  GetServerChanges  Module;
  SetHeartbeat      Module;

  I;
]

Init [
  If 1 Main;
  [
    {***** Register the sample service with RPC Manager. *****}
    RPCStatus    = \RPCManager\Register(SvcName, Invalid, Invalid,
                                        Invalid, \RPC_SYNC_MODE);
  ]
]

Main [
  {***** Maintain a couple of variables just to show the status of
the service.
         These can be displayed on a VTS page *****}
```

```
  ServiceMode = *RPCStatus == 2    { Server status == 2 }
                  ? "Server"
                  : *RPCStatus == 1 { Client status == 1 }
                    ? "Client"
                    : "Unknown"     { Unknown yet == 0   };
  CurrentServer = \RPCManager\GetServer(SvcName);

  If Watch(1, *RPCStatus);
  [
    IfElse (*RPCStatus == 2, Execute(
      {***** Just become server...set the output value to the last
received
             input value, plus 1 *****}
      HeartbeatCountOut = HeartbeatCountIn + 1;
      ServerEnable = 1;
    );
    { else } Execute(
      ServerEnable = 0;
    ));
  ]

  {***** While I'm server, periodically increment the heart beat
"out" counter
         and update all clients *****}
  If Timeout(PickValid(*RPCStatus == 2, 0) && PickValid(ServerEnable,
0), 1);
  [
    \RPCManager\Send(SvcName             { Service sending the
message  },
                     \RemoteGUID         { GUID defining app
},
                     0                   { Cut-off mode - NORMAL
},
                     1                   { Send to server flag - TRUE
},
                     Invalid             { Machine name or IP
},
                     1                   { Send to clients flag -
TRUE  },
                     0                   { Execute local flag - FALSE
},
                     1                   { Recursive flag - TRUE
},
                     "PeriodicRPC"       { target RPC module to
execute },
                     "SampleService"     { context to execute in
},
                     Invalid             { object for update caching
},
                     Invalid             { InputSessionID
},
                     HeartbeatCountOut++ { RPC module Parameters
}
    );
  ]
]
```

```
<
{================================ PeriodicRPC
================================}
{ Periodically RPC, called on the server and all clients, to update
the       }
{ heart-beat (synchronized state).
}
{=============================================================================-
=========}
PeriodicRPC
(
  Sequence                      { Info regarding call
};
)

PeriodicRPC [
  If 1;
  [
    CriticalSection(
      IfElse ((PickValid(HeartbeatCountIn, 0) != 1) &&
              PickValid(Sequence != HeartbeatCountIn + 1, 0),
        IfThen (PickValid(*RPCStatus != 2, 1),   { No sequence errors
on server. }
          SequenceErrors++;                      { It's already
bumped the count. }
        );
      { else } Execute(
        HeartbeatCountIn = Sequence;
      ));
      LastHeartbeatCount = Sequence;
    );
    Return(Invalid);
  ]
]

{ End of SampleService\PeriodicRPC }
>
```

Let us examine the code in a bit more detail:
The first task of the service code is to register itself as a service to RPC Manager:

```
    RPCStatus    = \RPCManager\Register(SvcName, Invalid, Invalid,
                                        Invalid, \RPC_SYNC_MODE);
```

The Register() call is passed the name of the service. This must be unique within the application, but can be the same name as a service within another application. The fifth parameter to Register() is the only other one we are using at present and is set to a special constant value \RPC_ SYNC_MODE which tells RPC Manager that our service has a syn- chronizable state. Omitting this parameter, or setting it to zero results in

a service that has no synchronizable state. In this case, there is no need to provide any of the synchronization support modules that we will see presently.

Register() returns a reference to a value that RPC Manager maintains on behalf of the service. The value that RPCStatus addresses (*RPCStatus) will be set by RPC Manager to one of three values:

- 2 – If the service instance is the server.
- 1 – If the service instance is a client.
- 0 – If the state of the service instance has not yet been decided.

The remainder of the code in state Main uses *RPCStatus to determine what to do. ServiceMode and CurrentServer are set to values that can be put on a display page. They are not necessary for service operation.

When the service instance becomes the server and on the first evaluation of the service main module, the line:

```
If Watch(1, *RPCStatus);
```

evaluates to TRUE. This provides a point at which initialization can be performed when the instance that is server changes.

Finally, a timeout statement trips once per second on the server, to broadcast the counter contents to all connected clients and then increment the counter. The ServerEnable variable is simply there to prevent a race condition that would arise with two statements watching the same data value (*RPCStatus).

The RPC issued by the Send() call invokes module PeriodicRPC() on the server and all clients, carrying the count as the sole parameter to this RPC subroutine. When PeriodicRPC() is running in the server instance, it does not check the sequence number for errors.

Note that PeriodicRPC() performs its work in a CriticalSection, as RPC subroutines are run on the RPC Manager thread, whereas the remainder of the service code runs on the service thread.

This prevents erroneous results caused by one thread modifying a value shared by the service and the RPC subroutine. An alternative technique is to have the RPC subroutine launch a module that interacts with the service's values. A module so launched, runs on the application thread.

Selection of the most appropriate method is usually determined by the complexity of the operations to be performed in response to an RPC sub-routine invocation.
Note also that the name for the service that is used for registration:

```
  SvcName = "Sample"            { Deliberately different from the mod-
ule name
```

is different from the module name. The two names can be, and usually are, the same. However, they are two different entities, the service name being used to identify the service within the application and the module name used to identify a module within the scope of \Code. The Send() call, in state Main, uses the module name to specify the context in which to find the RPC module, PeriodicRPC, but uses SvcName to identify the service within the application. The difference lies in the service name being used to determine which machines within the distributed system to send the RPC to, while the module name identifies the module within \Code that contains the specified RPC module name.

Related Information:

...Adding Server-Only Synchronization

...Configuring the Service

...Adding More Servers

...Server List Consistency

...Client Revision Information

...Client Changes

...Read and Write Locks

...Revised Code Example - revised for cross-application RPC

Adding Server-Only Synchronization

The code in the preceding topic is sufficient to perform the work of the simple service, but requires the addition of a pair of modules to source and sink the synchronizable state when so requested by RPC Manager:

```
  {***** Required Synchronization modules, which RPC Manager will
call *****}
  GetServerChanges  Module;
  SetHeartbeat      Module;
```

GetServerChanges() is launched by RPC Manager, on the server, when the service must provide its synchronizable state for a client. RPC Manager is careful to launch this module on the same execution thread as the service so that, while executing a script in GetServerChanges(), nothing else can execute on the service's thread. This can greatly simplify the acquisition of the synchronizable state. It is still possible, however, to have a launched RPC or a module launched by an RPC subroutine run during execution of a script in GetServerChanges(), as launched modules typically do not perform all their work in one script. If any such modules can modify the synchronizable state of the service, it is essential to either place the acquisition of the synchronizable state by GetServerChanges() within a CriticalSection, or to wait in GetServerChanges() until all such modules have completed execution.

On entry to GetServerChanges(), all service RPCs for the service are suspended (RPCs for other services and directed RPCs are still processed), allowing GetServerChanges() to wait for any launched modules that can modify the synchronizable state to finish. It is good design to only have your synchronizable state modified by RPC subroutines or modules launched by RPC subroutines. In this way, as no more service RPCs will be started until GetServerChanges() indicates that they can, you can guarantee that the synchronizable state of your service will not change while GetServerChanges() samples it.

GetServerChanges passes the service's synchronizable state to RPC Manager by returning a "packed stream" of RPC calls, which will be made on the client. This enables the re-construction of the synchronizable state to be achieved by making a sequence of RPC calls on the client, ensuring that the entire sequence of RPC calls is delivered to the client as one package, without scope for communication interruptions causing a partial update of the synchronizable state on the client. On the client, each component of the RPC package is executed in the strict sequence that

they are packed into the stream on the server. No other RPC call can interpose in this sequence on the client.
The second of the two modules is SetHeartbeat(). This is simply a service-specified RPC subroutine that, in our simple service case, receives, as a parameter, the synchronizable state of the service and stores it on the client. This RPC subroutine is the only RPC in the call package delivered from the server.
Let's look at the code for GetServerChanges():

```
<
{======================== GetServerChanges
==================================}
{ Called by RPC Manager during startup sync, on a server, to get the
package  }
{ of RPCs which create a synchronizable state on the client which is
in step  }
{ with the server.
}
{=================================================================-
=========}
GetServerChanges
(
  RevisionInfo                  { Revision info from GetCli-
entRevision call
                                  made on synchronising client
};
  PackStreamRef                 { Pointer to var to receive chnages
};
  ClientName                    { Name of client
};
  Guid                          { GUID of the syncing application...
};
  SyncMonitorObj                { Object value of my RPCMan-
ager\ServerSync.
                                  Goes Invalid if sync aborts...VTS
10.0 on   };
)

[
  Stream                        { Stream of changes to go to client
};
]

Sample [
  {***** This delay is not necessary. It is only here to prove that
our
         synchronizable state modification RPCs are suspended until
we
         call SetDivert. *****}
  If Timeout(1, 5) Wait;
  [
    {***** Sample the synchronizable state and build an RPC package
```

```
to send to
           the synchronizing client. *****}
    Stream = \RPCManager\PackRPC(Stream, "SetHeartbeat" { module },
                                 Invalid { scope },
                                 { Parameters: } Cast(LastHeart-
beatCount, 4));
    {***** Start diverting all RPCs for this client from here
onwards. RPC
           Manager will release the divert when synchronization done.
This
           also enables the flow of service RPCs for this machine
(the server).
           Expect a flood of 5 service modification RPCs to arrive!
*****}
    \RPCManager\SetDivert(SvcName, ClientName);
  ]
  {***** If we lost the synchronizing session, abandon all hope!
*****}
  If !Valid(SyncMonitorObj) Done;
]

Wait [
  {***** This delay is not necessary. It is only here to prove that
the service
         RPCs can continue once the RPC package has been built and
SetDivert
         called. At the end of this delay, this module will pass the
sampled
         state to RPC Manager which, in turn, passes it to the cli-
ent. Once again,
         if we lose the synchronizing session, abandon. ******}
  If Timeout(1, 10) || !Valid(SyncMonitorObj) Done;
]

Done [
  If 1;
  [
    *PackStreamRef = Stream;
    Slay(Self(), 0);
  ]
]

{ End of SampleService\GetServerChanges }
>
```

As stated above, RPC Manager launches this module when the server instance must provide the synchronized state of the service for a client. The parameters to GetServerChanges are:

RevisionInfo	See Client Revision Information.
PackStreamRef	A reference to a variable into which GetServerChanges () will store a packed stream of RPC calls that will be

	executed on the client.
ClientName	The workstation name of the client instance that is synchronizing with the server instance
Guid	The GUID of the synchronizing application. For cross-application RPC services, this will be different from the GUID of the application under which GetServerChanges runs.
SyncMonitorObj	The object value of the RPCManager object that is monitoring the synchronization sequence. If this goes Invalid, you should abort the current synchronization. Typically this will be because communications with the client have been lost. Only available on VTS 10.0 onwards.

The script in state Sample does the important work of this code. While this script is running, nothing else on the service's execution thread can run. The first job is to pack together all the RPC calls that have to be made on the client to generate the same synchronizable state as is present on the server. In our simple case, this consists of a single call to SetHeartbeat with the current count as its only parameter:

```
    Stream = \RPCManager\PackRPC(Stream, "SetHeartbeat" { module },
                                 Invalid { scope },
                                 { Parameters: } Cast(LastHeart-
beatCount, 4));
```

After the RPC package has been built into a temporary stream, the RPC Manager module SetDivert() is called:

```
    \RPCManager\SetDivert(SvcName, ClientName);
```

This causes two things to happen:

- All RPCs for the service that are destined for the synchronizing client are held in a queue by RPC Manager until synchronization is complete. The effect of this is to allow the server instance to continue to perform its normal work, including updating other clients, without the risk of updates arriving at the synchronizing client instance before the synchronizable state has been stored there.

- RPCs arriving for this service were held in abeyance when GetServerChanges () was launched. These RPCs are now permitted to flow.

The intent is that the synchronizable state of the service is now allowed to change. Changes to the state will be routed to in-sync clients and will be buffered for the synchronizing client until the client has processed the RPCs that bring it up to the same state as was sampled at the server.
Although the sample GetServerChanges shown is a launched module, (specifically to incorporate the code in state Wait to demonstrate the function of SetDivert), you can also write it as a subroutine module. In this case, it must return Invalid when it is finished its work. If you return a valid value, RPC Manager will hang. There is no particular advantage in choosing a subroutine over a launched module. You can simply choose the form that suits your needs best. The same is true of two other service synchronization modules, called by RPC Manager, named GetClientRevision and GetClientChanges. (Described later.)
The client stores the synchronizable state when RPC Manager unpacks the RPC package on the client. In this case, the only call in the package is to SetHeartBeat():

```
<
{============================== SetHeartbeat
==============================}
{ Called, on the client, by the RPC contained in the package gen-
erated by the }
{ server, during GetServerChanges.
}
{=================================================================-
=========}
SetHeartbeat
(
  Sequence                      { Info regarding call
};
)

SetHeartbeat [
  If 1;
  [
    HeartbeatCountIn = Sequence;
    \RPCManager\SetSyncComplete(SvcName, Invalid, 1);
    Return(Invalid);
  ]
]

{ End of SampleService\SetHeartbeat }
>
```

```
{ End of SampleService }
```

This simply stores the synchronizable state in HeartbeatCountIn and then informs RPC Manager that it is now synchronized. This is achieved by the call to SetSyncComplete(). It is vital that the client makes this call. Service synchronization will hang if this call is not made. Note that SetHeartbeat () does not have to make this call. It can be a separate call within the RPC package, but it must be made.
Once SetSyncComplete() is called, RPC Manager releases the queue of service requests (on the server) for this client and all subsequent service RPCs start flowing normally to the newly synchronized client.

Configuring the Service

Now we will incorporate our service into a VTScada application and configure it. First, create a new, standard VTScada application. Do not enable remote configuration yet. Set the start page title to be "Test Service" and the start page file name to be "TestServ".
We want create a VTScada page that displays the innards of our service, just so we can see what is going on. Edit the "Pages\TestSrv.SRC" file to be like the following:

```
[
  Title = "Test Service";
]

Main [
  Return(Self);

    { RPC status }
  \System\Edit(40, 110, 180, 70, "RPC Status:",
               \SampleService\ServiceMode, 0);

    { Server name }
  \System\Edit(230, 110, 370, 70, "Server:",
               \SampleService\CurrentServer, 0);

    { Sequence Error Counts }
  \System\Edit(40, 210, 180, 170, "Sequence Errors:",
               \SampleService\SequenceErrors, 0);

    { Sequence Numbers }
  \System\Edit(40, 260, 180, 220, "Sequence Act:",
               \SampleService\LastHeartbeatCount, 0);
  \System\Edit(230, 260, 370, 220, "Sequence Exp:",
```

```
                \SampleService\HeartbeatCount, 0);
]
```

A better way of doing this would be to interactively drop an instance of a widget that displays the required information, on a page. The creation of widgets is outside the scope of this document however, so the above will be sufficient for now.

Adding a module declaration to the VTScada application's AppRoot.SRC file incorporates the service into the application. Suppose that the service code above is contained in file SAMPSERV.SRC. Then the AppRoot.SRC may appear as follows:

```
[
  Constant POINTS        = 0x0002     { Point template class
};
  Constant GROUPS        = 0xFF00     { Collections of point types
};
  Constant LIBRARIES     = 0xFF01     { Library class module
};
  Constant GRAPHICS      = 0xFF02     { Shared widgets for points
};
  Constant PAGES         = 0xFF03     { Graphic pages & dialogs
};
  Constant SERVICES      = 0xFF04     { Service class
};
  Constant PLUGINS       = 0xFF05     { Plug-in class
};
  Constant PRIORITYSTART = 0xFF06     { Items that are pre-started
};
  Constant TSERVICES     = 0xFF07     { Threaded service class
};


  [ (POINTS)        {========= Modules that are point templates
===========}
  ]

  [ (GROUPS)        {==== Modules that are collections of point types
=====}
  ]

  [ (LIBRARIES)     {======== Modules that contain library objects
========}
  ]

  [ (GRAPHICS)      {= Modules that are shared widgets for points =}
  ]

  [ (PAGES)         {===== Modules that are graphic pages and dialogs
=====}
    TestSrv Module "Pages\TestSrv.SRC";
  ]
```

```
  [ (SERVICES)      {===== Modules that are services that are started
=====}
    SampleService Module  "SampServ.SRC";
  ]

  [ (PLUGINS)       {===== Modules added to other base system modules
=====}
  ]

  [ (PRIORITYSTART) {===== Modules/variables that are to be pre-
started ====}
  ]

  [ (TSERVICES)     {====== Modules that are threaded services to run
=====}
  ]
]
```

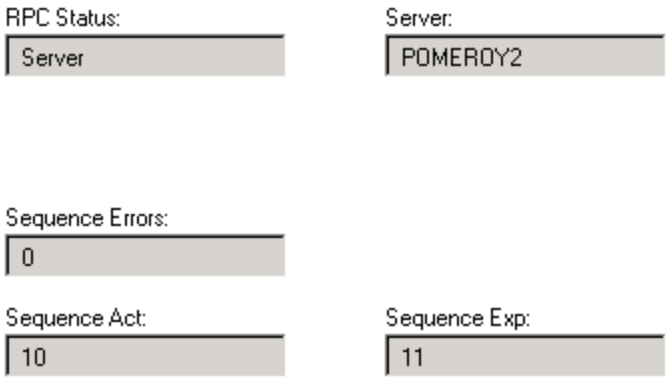
Note that VTScada has already added a similar line for the initial page that we added.

Before we can use the service, it is necessary to inform VTScada which machine, or machines are potential servers for the service. This is normally done in the application's Servers.XML file. A typical Servers.XML for our simple application might look like this:

```
<?xml version="1.0"?>
<RPC>
   <ServerLists>
      <Service Name="Default Server Lists">
         <Workstation Name="Default for Workstations">
            <Server>FREDSPC</Server>
         </Workstation>
      </Service>
   </ServerLists>
</RPC>
```

In our simple example, we have only one potential server, "FREDSPC", for all workstations and all services. All other machines that run this service will be clients to FREDSPC. You should use the workstation name of the machine that you wish to be the server, instead of FREDSPC. The application properties dialog provides the Edit Server Lists panel as a graphical interface for editing this XML file.

Now compile the application and run it. If all has gone well, you should see a display similar to the following: (navigation and title bar will vary by VTScada version)



Next we will enable remote configuration. That way, when you bring on one or more clients, the remote configuration service will ensure that subsequent code changes you make on the server will be correctly propagated. Start up the server instance again. On another machine, which will be a client, acquire and run the application. You should now have a server and a client that will maintain the simple numeric synchronizable state, no matter how you disrupt communication between them, or stop and start the application.
You can repeat the last step to add more and more clients, if you wish.

Adding More Servers

By simply changing the application configuration, you can add more potential servers to the list for our simple service. This enables you to experiment with "fail-over" from one server to another. Client machines will automatically re-synchronize to whichever machine is currently the highest available server.
For example:

```
<?xml version="1.0"?>
<RPC>
   <ServerLists>
      <Service Name="Default Server Lists">
         <Workstation Name="Default for Workstations">
            <Server>FREDSPC</Server>
         </Workstation>
      </Service>
      <Service Name="Sample">
         <Workstation Name="Default for Workstations">
            <Server>FREDSPC</Server>
            <Server>JOESPC</Server>
```

```
            <Server>GONZOSPC</Server>
          </Workstation>
        </Service>
    </ServerLists>
</RPC>
```

In this configuration, the Sample service has three potential servers, FREDSPC, JOESPC and GONZOSPC arranged in order of decreasing priority. You may have noticed that a different section was used for this (all other services are using the default list with just the single server, FREDSPC). This enables you to specify different server lists for different services.

The XML Service tag specifies potential server lists for the specific service identified by the Name property, with the "Default Server Lists" Name specifying the potential server list for all services that do not have their own potential server list section.

Now, the machine nearest the top of Sample's server list, which has an available service instance, will be the current server for the service. If a higher-ranking machine becomes available, the higher-ranking machine will synchronize with the current server instance and then seize server status from the lower ranking, which will then switch into client mode and synchronize with the new server. All client machines will, likewise, synchronize to the new server.

By now you may have realized that this architecture is distinct from the traditional server/client architecture, where one or two machines were designated as server. Typically, one machine would be the "hot" standby for the other machine. The VTScada distributed architecture enables different services to have their server instances executing on different machines, each with different fail over strategies. The VTScada distributed architecture enables you to focus on writing solutions for your services, with minimal coding overhead to support such a versatile architecture.

The example is insufficient for some, more sophisticated needs. The following sections will examine the extensions to the above that are provided.

Server List Consistency

It is important to ensure that each machine that can be a server for a service has a consistent list of server machines. Failure to observe this will result in sporadic service synchronization failure. Note the wording of this statement. A machine that can never be a server, i.e. can only be a client can have a different server list from the server. This is the method used to implement clients-of-clients. However, a machine that can be a server must have a consistent list of servers specified for that service. Once again, note the careful wording. Consistent does not mean identical, but if a server has a different list of machines from another server, they are only allowed to be different by omitting some machines from the tail of the list.
For example, the server lists for the Sample service could be configured as follows:

```
<?xml version="1.0"?>
<RPC>
   <ServerLists>
      <Service Name="Default Server Lists">
         <Workstation Name="Default for Workstations">
            <Server>FREDSPC</Server>
         </Workstation>
      </Service>
      <Service Name="Sample">
         <Workstation Name="Default for Workstations">
            <Server>FREDSPC</Server>
            <Server>JOESPC</Server>
         </Workstation>
         <Workstation Name="GONZOSPC">
            <Server>FREDSPC</Server>
            <Server>JOESPC</Server>
            <Server>GONZOSPC</Server>
         </Workstation>
      </Service>
   </ServerLists>
</RPC>
```

You may wish to use such a configuration if machine GONZOSPC was in a remote location from FREDSPC and JOESPC. Then, if the communication medium between the two locations is broken, the area without GONZOSPC [the "Default for Workstations" list for the "Sample" service, used by FREDSPC and JOESPC] will use the higher available server of FREDSPC and JOESPC, whereas the area with GONZOSPC will lose

communication with FREDSPC and JOESPC and so will fall back to GONZOSPC.
Of course, precisely the same effect would be obtained by having each machine use the first list, but this method reduces communication traffic between the two areas slightly, by not having to have FREDSPC and JOESPC examine the server status of GONZOSPC. It also prevents FREDSPC and JOESPC from every synchronizing to GONZOSPC. If GONZOSPC is isolated from the other two, higher servers, it will re-synchronize to the higher order servers when communication is restored, destroying the local state of the service on GONZOSPC.

Client Revision Information

Our simple service had very little data in its synchronizable state. A service will typically have a much richer set of data. While a set of packed RPCs, along with the ability to pass streams as RPC parameters, are a very flexible means to cope with diverse sets of data, some data sets may be so large that it is desirable to minimize the amount transferred. Typical data sets that fall into this category include data that is acquired over a period of time, but has to be retained. The classic example of this is alarm history data. A user may wish to display such data on a client machine at any time. The programmer is left with an unenviable choice:

1. Transfer all data during service synchronization. This lengthens synchronization times significantly and becomes impractical if many clients are attempting to synchronize concurrently. It has the advantage that, after synchronization, response times to user requests are short, as the data is local to the client.
2. Leave all the infrequently accessed data on the server. This avoids the lengthy synchronization times, but gives a poorer response time to user requests to access such data.

RPC Manager provides support for a compromise between the two extremes, by transferring only that part of the data that the client doesn't have. Initially, the client has to be passed the entire set of synchronizable data, but subsequent synchronizations are much briefer, as

only the data that the client doesn't have is transferred. Such an arrangement implies that the client must have the ability to retain the bulk of the synchronizable state on non-volatile storage.

You may have noticed that the GetServerChanges() module that was developed in the Simple service example did not make any use of its first parameter, the RevisionInfo parameter. That parameter is a stream which RPC Manager obtains by calling GetClientRevision() on the synchronizing client.

GetClientRevision() is provided by the service code and RPC Manager passes in a reference to a variable which it expects to be set to the stream of client revision information. The contents of the stream is entirely service specific and contains whatever revision information is appropriate for the service to represent the latest data that the client instance has. RPC Manager places no interpretation on the contents of the stream. Typically, GetClientRevision() will populate the stream with one or more VTScada timestamps.

```
<
{======================== GetClientRevision
==================================}
{=============================================================================-
=========}
GetClientRevision
(
  RevisionRef                   { Reference to revision information
};
)

Only [
  If 1;
  [
    *RevisionRef = BuffStream("");
    {***** Populate the stream with a single timestamp *****}
    SWrite(*RevisionRef, "%5b", LastKnownTimeStamp);
    Return(Invalid);
  ]
]

{ End of GetClientRevision }
>
```

It then becomes the server instance's responsibility to utilize the information within the RevisionInfo parameter to GetServerChanges() to gen-

erate the correct RPC package to bring the client up to date with the server.
GetClientRevision() is launched by RPC Manager on the service thread. Note that this module can be either a subroutine, returning Invalid, or a launched module that slays itself when complete. In the example above, we have chosen to make it a subroutine module.

Client Changes

There is one more method that a service can provide that RPC Manager uses during synchronization – GetClientChanges(). In the same way that the server instance builds an RPC package for execution on a client instance, the client instance can provide an RPC package for execution on the server. The RPC package from the client is executed after GetServerChanges() has been called. This ensures that any changes that result from the client instance's RPC package being executed are not reflected in the RPC package generated by the server instance. Any changes in the synchronizable state of the service that result from executing the client instance-provided RPC package will cause normal service RPCs to be made to update all client instances.
It is uncommon for this facility to be required, however there are some circumstances in which it is necessary. The usual reason is where the client instance can operate independently from the server instance, if it loses the ability to communicate with the server instance.
It is essential for SetSyncComplete() to be called by the server instance, in the process of executing the RPC call stream that the client instance generated, just as it was essential for the client instance to call SetSyncComplete() during execution of the server generated RPC package. Once again, SetSyncComplete() can be called either by the last RPC subroutine in the package or the SetSyncComplete() can be embedded within the package.
The latter technique enables the RPC subroutines to be used other than for purely synchronization, by not embedding the SetSyncComplete() in an RPC subroutine. The following code demonstrates this:

```
<
{============================== GetClientChanges
==========================}
{ This module returns the changes seen on the client since the server
was   }
{ last connected.
}
{=============================================================================-
=======}
GetClientChanges
(
  PtrPackStream               { Pointer to var to receive changes
};
)

[
  Stream                      { Stream of changes to go to client
};
]

Only [
  If 1;
  [
    Stream = \RPCManager\PackRPC(Stream, "SomeRPC" { module },
                                 SvcName { scope  },
                                 { Parameters: } 69);
    Stream = \RPCManager\PackRPC(Stream, "SetSyncComplete" { module
},
                                 "RPCManager" { scope  },
                                 { Parameters: } SvcName, Invalid,
1);
    *PtrPackStream = Stream;
    Slay(Self, 0);
  ]
]

{ End of GetClientChanges }
>
```

RPC subroutines that have a need to know whether they are being called via RPC or directly can determine this by comparing the current thread name against the name of the RPC Manager thread, as follows:

```
  IfThen (ThreadName(Self()) == "RPC",
    { Running on the RPC Manager thread }
  );
```

Related Functions:

... GetServerChanges

... SetSyncComplete

... PackRPC

Read and Write Locks

The simple service that we have developed has the luxury of being able to acquire its synchronizable state within one script in the service instance. This guaranteed that the synchronizable state could not change while it was being acquired for transmission to the synchronizing client. Once the SetDivert() call has been made, the service can modify the synchronizable state, safe in the knowledge that updates for the synchronizing client will be held in abeyance until the client has processed the RPC package.
However, if it is not possible to acquire the synchronizable state within a single script, some form of semaphore is going to be required, to hold the service off from modifying the synchronizable state until it has been sampled. Such a situation may arise if part of the synchronizable state was held on non-volatile storage, or if the VTScada statements needed to acquire the synchronizable state could not be used in a script.
RPC Manager provides such a semaphore and maintains one per service instance. The semaphore provides two types of "lock" which can be requested and acquired, a "Write" lock and a "Read" lock. These behave according to the following two rules:

- Any number of Read locks may be granted simultaneously but they will not be granted if a Write lock exists.
- Only one Write lock may be granted at a time and it will only be granted if there are no Read locks.

The idea is that a service module that wishes to examine the synchronizable state requests a Read lock, whereas a service module that wishes to modify the synchronizable state requests a Write lock. This enables many service modules to concurrently examine the service data without fear of it changing.
RPC Manager automatically acquires locks for a service during the synchronization process, as follows:

- A Write lock is obtained on the client at the start of service synchronization. It is released when SetSyncComplete() is called on the client or when the synchronization process is aborted by RPC Manager, due to communication

failure with the server instance.

- A lock is obtained on the server immediately before GetServerChanges() is called. If the client instance provided an RPC package, a Write lock is obtained, otherwise a Read lock is obtained. The lock is released when the client has processed the RPC package from GetServerChanges() and the RPC package from the client instance (if any) has been processed, or when the synchronization process is aborted by RPC Manager, due to communication failure with the client instance.

Note that if you choose to use locks, you must be very careful to protect all methods which access the synchronizable state by the correct locking call:

```
\RPCManager\ReadLock(&Ready, SvcName);
\RPCManager\WriteLock(&Ready, SvcName);
```

Both lock calls require a reference to a variable as their first parameter. The variable value will be set to one when the lock has been obtained. The service name is passed as the second parameter. To release the lock, simply stop, or slay, the ReadLock() or WriteLock() instance that the above statements are running.

Synchronization Sequence

Putting together all the parts above, the full synchronization sequence is:

1. RPC Manager obtains a Write lock on the client.
2. When the Write lock has been obtained, RPC Manager launches GetClientRevision() on the client, into the service scope and on the service's thread.
3. RPC Manager launches GetClientChanges() on the client, into the service scope and on the service's thread. This is done after 2, but RPC Manager does not wait for GetClientRevision to complete before launching GetClientChanges().
4. RPC Manager then waits for both GetClientRevision() and GetClientChanges() to terminate, by slaying themselves or returning Invalid and then sends the client revision and changes streams to the server.
5. RPC Manager, on the server, obtains either a Read or a Write lock, depending on whether there is a client changes stream.

6. When the lock has been obtained, RPC Manager, on the server, atomically starts buffering service RPCs for the service and launches GetServerChanges ().
7. When GetServerChanges() calls SetDivert(), the buffered service RPCs are released for processing and service RPCs destined for the synchronising client start buffering.
8. When GetServerChanges() terminates, by slaying itself or returning Invalid, RPC Manager queues the RPC package from GetServerChanges() for transmission to the client. If an RPC package was received from the client, RPC Manager, on the server, now processes that call package, calling the RPC subroutines within the package and then waits for SetSyncComplete() to be called on the server.
9. RPC Manager, on the client, executes the RPC package from the server and waits for the client service instance to call SetSyncComplete(). When this is called, the client instance is marked as ready and the Write lock, obtained in 1, is released.
10. When the server sees the client instance marked as ready (and 7, above, has completed), any buffered RPCs from the service for that client are queued for transmission to the client and synchronization terminates, clearing the Read or Write lock, obtained in 5, above.

Related Information:

...Alternate Status - when and how to configure.

...Sticky Status

...Preventing Synchronization with Lower-Order Servers

...Server Evaluation Rules - for use with Alternate Status.

Alternate Status

The prioritized list of servers that are specified in a configuration file contains sufficient information to determine what to do when communication between servers or between servers and clients fails. This could be due to failure of the communications medium or failure of the software on a machine (perish the thought) or failure of the computer hardware.

There may, however, be good reason to be able to inform RPC Manager of the health of a service. Typically, this will be where a service has to communicate to, or use, some third-party equipment. The service will need to indicate that it is, or is not, capable of performing the functions that it was designed for. A driver may typically use this facility to show its ability to communicate with a PLC.
This is achieved by setting or clearing the Alternate status of the service. RPC Manager monitors the Alternate setting to determine the most suitable service instance for the service. The rules by which server status is determined are described in the section on Server Evaluation Rules.

Related Information:
Server Evaluation Rules

Sticky Status

A service can indicate to RPC Manager that server status is to "stick" with the machine that currently has server status for that service, rather than migrating server status to the highest ranking available server machine. This feature is of use where to change over to another machine carries an unacceptable penalty. For example, a service that communicates with some external equipment may have to drive signals or issue protocol to perform a physical changeover of communication routing equipment. The system designer may wish this to only happen infrequently.
A service is configured to be sticky by specifying a "Sticky" setting in the Servers.XML file, as in the following example, which specifies the Sample service as sticky:

```
<?xml version="1.0"?>
<RPC>
   <ServiceSettings>
      <Service Name="Sample">
         <Setting Name="Sticky">1</Setting>
      </Service>
   </ServiceSettings>
   <ServerLists>
      <Service Name="Default Server Lists">
         <Workstation Name="Default for Workstations">
            <Server>FREDSPC</Server>
         </Workstation>
      </Service>
      <Service Name="Sample">
```

```
         <Workstation Name="Default for Workstations">
            <Server>FREDSPC</Server>
            <Server>JOESPC</Server>
            <Server>GONZOSPC</Server>
         </Workstation>
      </Service>
   </ServerLists>
</RPC>
```

If the Sticky setting is set to other than its default of zero, the service becomes a "sticky" service. If a sticky service instance that is the current server wishes to yield server status, it should set alternate status, see section Alternate Status. This will cause immediate migration of server status to the highest available server. The rules by which server status is determined are described in the section on Server Evaluation Rules.

Preventing Synchronization with Lower-Order Servers

There are some unusual circumstances where you may not wish a higher-order service instance to synchronize with a lower-order service instance that holds server status. Typically, one machine would be regarded as holding the master set of service data, regardless of whether a service instance was running on that machine or not. VTScada Security Manager is just one such service.
When initializing or on change of server, servers of a lower-order than the current server will synchronize with the current server. Servers of a higher-order than the current server will not synchronize with the current server.
To achieve this, the service must pass a valid non-zero value in the PrioritySync parameter to the \RPCManager\Register call, see section Service Control Methods.
This does not change the server evaluation rules or the synchronization sequence. It merely prevents synchronization to a lower-order server.

Server Evaluation Rules

The rules by which the server is determined encompass the prioritized list of servers specified in the application's configuration, whether or not

a service instance is synchronized, the Alternate status of a service instance and the Sticky status of a service instance.
Note that, in the rules that follow, the phrase "available service instance" means that a service instance has been registered and, if necessary, synchronized.

1. When a service registers with RPC Manager, it defaults to having its Alternate status clear. The prioritized list of severs is read from the application's configuration.
2. If a service instance sets itself into Alternate status the next available service instance that is not Alternate, down the prioritized list, is selected as the server for that service.
3. If all the service instances are in Alternate status, they are all forced out of Alternate status, except the service instance that was the most recent server, which remains in Alternate status. The highest priority available service instance is selected as the server.
4. If a service instance clears its Alternate status (or has it forcibly cleared by 3) then it is selected as the server only if there are no higher priority service instances available.
5. If there are no available service instances, the highest priority registered, but not synchronized, service instance is selected as the server.
6. If the selected server is not the current server and the current server is not a "sticky" server, the current server is demoted from server status.
7. The selected server is promoted to be the current server.
8. If there is only one server on the prioritized list for the service, neither Alternate status nor Sticky has any effect.

RPC Call-Backs

Certain events that happen within the lifetime of a service instance may be of interest to the service instance or to the application as a whole. RPC Manager looks for the existence of a module called RPCServerNotice () in the service code and in \Code for the application. If it finds such a module, it is called, as a subroutine when the following events occur and with the listed notification code, supplied as a parameter:

Notification Code	Reason
\RPCManager\#RPCCallbackRegistered	The local service instance has registered with RPC Manager.
\RPCManager\#RPCCallbackSessionClosed	A session with a remote service instance has been disconnected. The disconnection may or may not be the result of application shutdown..
\RPCManager\#RPCCallbackSessionOpen	A session with a remote service instance has been established.
\RPCManager\#RPCCallbackUnRegistered	The local service instance has unregistered with RPC Manager.
\RPCManager\#RPCCallbackSocketOK	A connection to a remote machine has been established (\Code call-back only).
\RPCManager\#RPCCallbackSocketLost	A connection to a remote machine has been lost (\Code call-back only).
\RPCManager\#RPCNewSessionData	A session table has been received from the remote system – this is synonymous with a socket connection being established. The RPC session tables will not yet be set up, so use of this call-

back should be restricted to detecting a new connection.

RPCServerNotice() is called with the following parameters:
RPCServerNotice(NotifyCode, IP, Name, LocalIP, LocalName, WorkstationName, ShortName, SameAsLast);
Where:

NotifyCode	One of the codes listed above.
IP	The IP address of the remote machine. For codes #RPCCallbackRegistered and #RPCCallbackUnRegistered this value is undefined.
Name	The name of the remote machine. For codes #RPCCallbackRegistered and #RPCCallbackUnRegistered this value is undefined.
LocalIP	The IP of the local machine. On a multi-homed network this could be any of the legal IP addresses by which this machine is known.
LocalName	The name of the local machine.
WorkstationName	The name of the local machine, as determined by a WkStaInfo(0) statement. This is normally the same as the previous parameter.
ShortName	Valid for \Code call-backs for service instance related events only. The name of the service to which the event relates.
SameAsLast	Valid for \Code call-backs for service instance related events only. Non-zero if this notification is not the same as the previous notification for the same machine.

Connection Configuration and Management

This section describes how RPC Manager maintains the inter-machine connections in a distributed system. It is provided as an understanding of the sequence of operations in RPC Manager makes for easier configuration and diagnosis of problems.

Related Information:

...Link Maintenance Cycle - when and how links are created.

...Multi-homed Systems - making use of multiple network interfaces.

...Clients of Clients - provides the ability for some machines to maintain different server lists

...WANs - support for.

Link Maintenance Cycle

A machine will not attempt to open a connection to another machine until it has a reason to do so:

1. A service instance has registered and needs to communicate to a machine.
2. A directed RPC is made to the remote machine.
3. RPC Manager is explicitly requested to open a connection to a machine.
4. The remote machine initiates communication, due to one of the above reasons.

When a connection is to be established for the first time, RPC Manager creates a MachineNode to oversee communication with the remote machine and one or more SocketNodes to maintain the raw socket stream connection between the machines.

The SocketNodes maintain the communication pathways between the MachineNode at either end of the link and report status changes to their owning MachineNode. The MachineNodes for the various interconnected machines report status changes that they cannot transparently manage to the \Code level RPCServerNotice(), (see the section RPC Call-Backs), and to the TagNodes that are using them. TagNodes use these events in their driving of service instances.

SocketNode\SocketOpen will, initially, be Invalid. The SocketNode will aggressively try to establish a socket stream connection with a SocketNode on the remote machine up to the number of times specified in the RPCSktConnectAttemptMax system configuration setting, before deciding whether to mark the SocketNode open or not. If it fails to obtain a satisfactory connection within this number of attempts, SocketNode\SocketOpen is marked as closed (0) and MachineNode informed. Once a socket stream has been opened, SocketNode interrogates the stream to get detailed network information. It then marks the SocketNode open (SocketOpen>0) and informs MachineNode.

Session management information is now retrieved from MachineNode and transmitted to the remote MachineNode. When the MachineNodes first successfully exchange session management information, each of the two MachineNodes is marked as open (MachineNode\SocketOpen > 0). At this point, "pinging" is initiated down the socket stream. A "ping" is just a "keep-alive" transmission that elicits no response from the other end. Pings are very efficient and are only sent if there has been no other communication activity on the socket stream for the number of seconds defined in the RPCPingInterval system configuration setting.

If no activity is observed on a socket stream for the number of seconds defined in the RPCReconnectTime system configuration setting, the SocketNode terminates the socket stream and attempts to open another one. Once again, RPCSktConnectAttemptMax attempts are made at establishing a connection during the connection establishment process, as described above.

If, after RPCReconnectTime expired, no connection can be established within the number of seconds defined in the RPCSocketDeadTime system configuration setting, any sessions which the MachineNode has with applications running on the remote machine are terminated and their session IDs set to RPC_NO_SID. The session IDs will be set to new values when the session tables are re-built after a connection is re-established with the remote machine.

MachineNode will attempt to transmit RPCs to the remote machine while it is marked as open. Each RPC transmitted receives positive acknowledgment of its receipt. Failure to receive this acknowledgment within the number of seconds defined in the RPCResendDelay system configuration setting causes MachineNode to resend it. There will be a maximum of RPCSktResendAttempts to transmit the RPC successfully. If, after these retries, the RPC has still not been acknowledged, the socket stream is terminated and SocketNode goes through its link re-establishment cycle, as above.
All RPCs carry with them an identification that prevents a re-transmitted RPC from being executed if it was received and executed, but the acknowledgment was lost.

Related Information:

...Link Tolerances

...Application Settings for RPC - reference for the properties mentioned here.

Link Tolerances

When two machines connect, the response time from one machine to another can be variable, partly due to CPU loading, but mostly due to link throughput limitations. To address this, a set of link "tolerances" can be specified in the SETUP.INI file. The tolerances are only applied to the RPCResendDelay, RPCReconnectTime and RPCPingInterval settings. The RPCPingInterval is only affected on the receiving machine when it is used to calculate the connection timeout.
A "tolerance factor" is specified as a percentage value, so a tolerance factor of 100 is unity, 300 is 3 times the value and 50 is half of the value. This enables the specification of a base, system default, set of values and tolerances for each peer-to-peer connection that requires them.
It is desirable to be able to specify the settings for an entire system in one file [SETUP.INI], which can then be installed on all machines in the system. This is achieved by having an INI file section heading which

identifies a particular machine or IP interface in the system and having all the tolerance settings [for other machines with which that machine can communicate] specified as entries within the section. There are, however, a number of variations in how this is specified. On a system consisting entirely of single homed machines, "LinkTolerance-Name" [where name is a workstation name, as returned by WkStaInfo()] is sufficient to uniquely identify the network pathway from that machine to other machines. For multi-homed systems, it may be necessary to use "LinkTolerance-IP" [where IP is the IP of a network interface] to identify tolerances for different network pathways between two machines.
To resolve ambiguity and provide flexibility, each machine uses the following methods to extract the tolerance setting from this .INI file, when a connection is established. The rules are performed in the order listed:

1. A search is made for a section [LinkTolerance-LocalIP], where LocalIP is the IP of the local machine. If found, that section is searched for the IP of the remote machine. If the IP of the remote machine is not found, the section is searched for the name of the remote machine.
2. If 1 did not produce a tolerance, a search is made for a section [LinkTolerance-LocalName], where LocalName is the name of the local machine. If found, that section is searched for the IP of the remote machine. If the IP of the remote machine is not found, the section is searched for the name of the remote machine.
3. If 2 did not produce a tolerance, a search is made for a section [LinkTolerance-RemoteIP], where RemoteIP is the IP of the remote machine. If found, that section is searched for the IP of the local machine. If the IP of the local machine is not found, the section is searched for the name of the local machine.
4. If 3 did not produce a tolerance, a search is made for a section [LinkTolerance-RemoteName], where RemoteName is the name of the remote machine. If found, that section is searched for the IP of the local machine. If the IP of the local machine is not found, the section is searched for the name of the local machine.

5. If 4 did not produce a tolerance, a search is made for a section [LinkTolerance-LocalIP], where LocalIP is the IP of the local machine. If found, that section is searched for an entry called "Default".
6. If 5 did not produce a tolerance, a search is made for a section [LinkTolerance-LocalName], where LocalName is the name of the local machine. If found, that section is searched for an entry called "Default".
7. If 6 did not produce a tolerance, a search is made for a section [LinkTolerance-RemoteIP], where RemoteIP is the IP of the remote machine. If found, that section is searched for an entry called "Default".
8. If 7 did not produce a tolerance, a search is made for a section [LinkTolerance-RemoteName], where RemoteName is the name of the remote machine. If found, that section is searched for an entry called "Default".
9. If 8 did not produce a tolerance, default to 100% of the tolerances.

Note that this procedure is only carried out the first time that a connection is established between two machines.

The "Default" setting within the sections allow you to specify a default tolerance for all connections from the machine or interface identified in the section header. This can significantly reduce the maintenance overhead of these tolerance sections.

For example:

```
[LinkTolerance-192.168.5.42]
  Default = 200

[LinkToleance-GONZOSPC]
  192.168.5.50 = 300
  Default = 150
  JOESPC = 100
```

This will cause machine 192.168.5.42 to have a 200% tolerance of all machines connected to it. GONZOSPC will have a 300% tolerance of machine 192.168.5.50, a 100% tolerance of JOESPC and a 150% tolerance of all other machines.

Note that these settings are specified in SETUP.INI and not in service configuration files.

Note: You use workstation names other than IP addresses whenever possible, especially in networks where the workstations may be dual-

homed, or where dynamic IPs are assigned. Since workstation names are unique, it is good practice to condition yourself to use workstation names, rather than IP addresses, even if your system configuration does not require the use of workstation names.

Multi-homed Systems

RPC Manager will take advantage of multiple network interfaces on a machine. Each network interface must have its own IP address.
There are two reasons why you might want to use a multi-homed system:

1. To improve the tolerance of the distributed system to network failures.
2. To separate the SCADA system network usage from a network used by business systems. The usage of a network by business systems can be quite unpredictable and dedicating a network to the SCADA and associated control systems may be prudent.

These two justifications have an area of overlap, in that the VTScada distributed system can see all networks connected to it, regardless of purpose.
Through configuration parameters, you can configure RPC Manager to use the available networks in two ways:

1. Prioritized. The available networks are arranged in an order of priority. RPC Manager always uses the highest priority available network. This could be used for either of the above two scenarios.
2. Round-Robin. All network interfaces are treated equal and RPC Manager traffic is sent using each interface in turn. This could only be used for scenario 1 and is the default behavior.

The two methods can be combined, where there are three or more networks, such that one of the networks could be prioritized and the other two left as round robin. RPC Manager traffic will always be sent over the prioritized network first, using the remaining two networks in round-robin mode if the prioritized network link to a machine fails.
For example:

```
[RPCManager-NetPriority]
  IP = 192.168.1.0/24
  IP = 192.168.2.0/24
```

Each line of the "RPCManager-NetPriority" section specifies an IP mask, of the form:

```
<IP address>/<number of bits>
```

such that the number of bits specified is applied to the IP address from the most significant end. A mask value of 24 specifies the first three numeric parts of the IP address (8-bits per number), a mask of 16 the first two numeric parts and so on.
The example above specifies that IPs on subnet 192.168.1 are to be given priority over IPs to the same machine on all other subnets. Likewise, IPs on subnet 192.168.2 are to be given priority over IPs to the same machine on all other subnets, except subnet 192.168.1. If an IP on subnet 192.168.1 becomes unusable, the corresponding one on subnet 192.168.2 will be used. If that also fails, any remaining IPs on other subnets will have RPCs transmitted to them in round-robin fashion.

Clients of Clients

It is common sense that a service instance should always see the same prioritized server list as all other service instances in the system, otherwise unexpected results may occur. However, it is deliberately possible to configure the distributed system in such a way that one (or more) machines can see different server lists. Such a situation may arise if a Remote Access Service (RAS) were to be configured so that the RAS client did not have free access to the LAN on which the distributed system is running. This may be desirable for security purposes, especially if the RAS connection is a dial-up.
In such a case, it is necessary to configure VTScada running on the RAS client as having the RAS server as its sole server instance. RPC Manager will synchronize the service instances running on the RAS client to those running on the RAS server. However, individual services may not provide full support by, for example, not broadcasting updates recursively to clients.

If the machine that is acting as both client and server is obliged to re-synchronize with its server, all its clients will be forced to re-synchronize with it, after it has synchronized with its server.

Related Information:

See the Configuration section for details on how to configure this.

WANs

Wide area networks (WANs) are supported indirectly, by the operating system support provided at the network interface level. The primary considerations when having a distributed system operating in a wide area domain are:

1. There will be reduced throughput and increased latency.
2. Security measures, e.g. firewalls, are in common use. Network administrators must open up the port required by RPC Manager for this to function. The port required is set in RPC Manager configuration, (see Configuration)

Configure Cross-Application RPC

This section discusses how cross-application RPC works and the modifications you will need to make to your application to achieve this.
Each application has it's own Globally Unique Identifier (GUID), which is generated by VTScada when the application is created and is stored in the SETTINGS.STARTUP file in the application directory.
In the scope of \Code, the VTScada loader creates two variables for your application, LocalGUID and RemoteGUID, both of which are preset to the application GUID. As you will see from the Application Control of Servership
In normal operation, RPCManager controls which service instance is currently the server for a service. The system is simply configured to instruct RPCManager how its servership control algorithms should operate.

In rare cases, it may be necessary to control the servership of one or more services by the application itself. Consider, for example, a system where maintenance of the system is necessary while the application is running. For operational reasons, the system owners do not want a number of key services to have servership held by the machine undergoing maintenance, but do want the application to be running.
In such circumstances the application needs to either be able to disable a particular machine from owning servership of the key services or, more generally, be able to control which machine is the server for those services. More complex situations could arise where, for example, a specific machine does not want to be considered a candidate for servership.

Related Information:

...Cross-Application Services - described.

...Cross-Application Service Variations - alternate configurations.

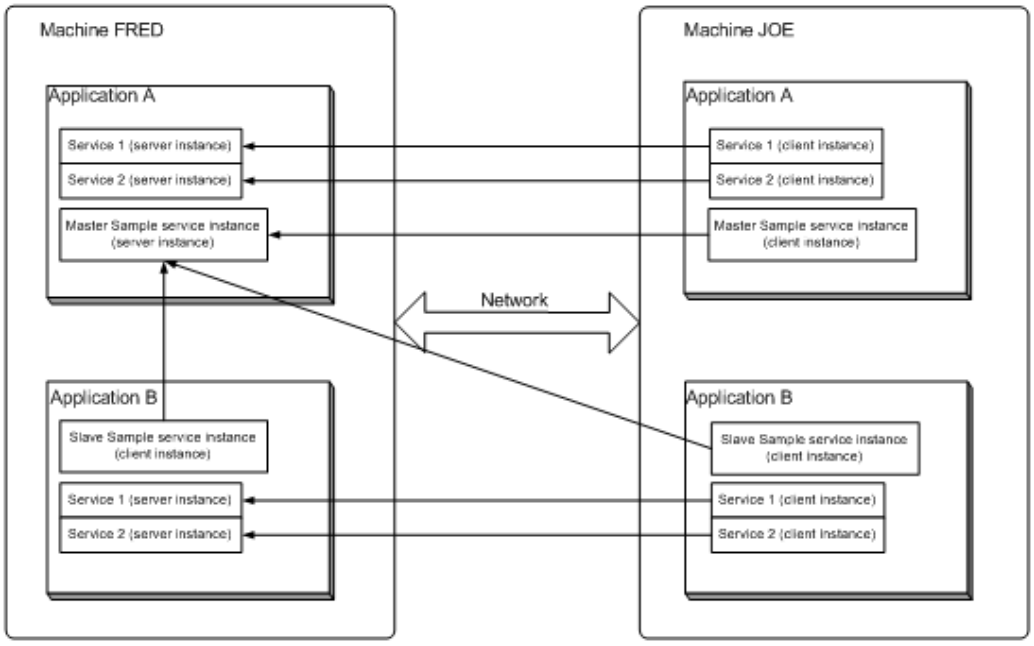
...Revised Code Example

...CurSourceAppGUID - for the rare case where the master (or a slave) needs to know which application is making an RPC to it.

Cross-Application Services

While cross-application RPC can be achieved between script applications it is in standard applications that it is most useful. In standard applications, RPCs are performed between services. In cross-application RPC in standard applications, services of the same name in different applications are maintained in synchronization.
One application houses the master service and the other applications house the slave services. The application housing the service with identical RemoteGUID and LocalGUID variables is the master. All other applications house slaves. There can be many different applications housing slaves, but only one application housing the master.
This terminology (master and slave) has been selected to try and avoid confusion between the terms server and client. In cross-application RPC, we define the term "server" consistently with non cross-application

services, as being the only service instance that has servership of the service. By definition, this is the master service.
Therefore, exactly one instance of the master service can be a server. All other instances of the master service are clients. All instances of the slave services are clients to the current master service server. This is depicted in the following diagram, where each arrow points to the server instance.



Each application has a service named "Service 1" and a service named "Service 2". These two services are completely independent between the applications, i.e. Application A's Service 1 has no relationship with Application B's Service 1. Service Sample, however, is a cross-application service. Application A houses the master service. Application B houses a slave service. Application A on machine FRED is the current server instance of this service. If you were to stop Application A on machine FRED, servership of Application A would transfer to machine JOE (on the assumption that the server list for all Application A services is FRED, followed by JOE). Application B's Sample service will follow the master service and both application B's Sample service instances will become clients to Application A's Sample service on machine JOE.

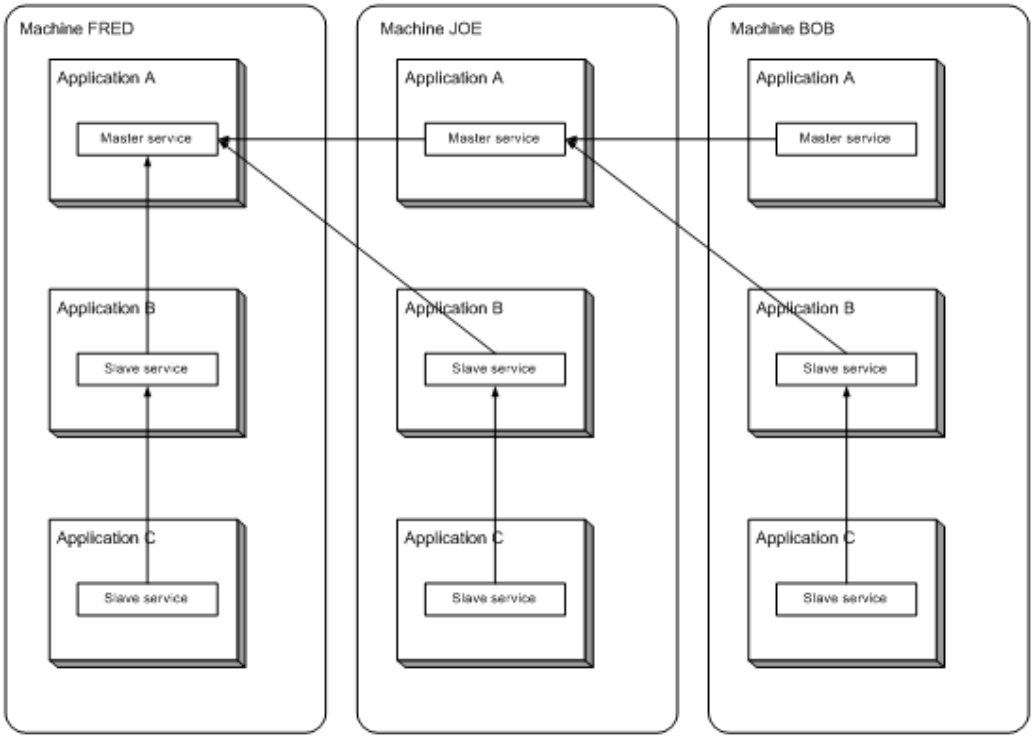
Note that Application B's Sample service can not be a server instance, hence the term "slave".

Related Information:

Cross-Application Service Variations - alternative arrangements.
CurSourceAppGUID - for the rare case where the master (or a slave) needs to know which application is making an RPC to it.

Cross-Application Service Variations

You can have many applications which house a slave service, for example, application A can house master service Sample. Applications B, C and D can all house slave services of Sample.
You can also have "slaves of slaves", in a similar manner to "clients of clients", the primary difference being that, because the slaves are in separate applications from the master, they can exist on the same machine (but don't all have to).
To configure a "slave of a slave" configure an intermediate application service instance (the slave) to be the master for a second application. For example, if application A (GUID GA) houses the master service Sample, application B (GUID GB) houses a slave service Sample and application C (GUID GC) houses the slave of slave Sample, then create a RemoteGUID variable in service Sample of application B and set it to GUID GA and create a RemoteGUID variable in service Sample of application C and set it to GUID GB. Now, B's Sample service will synchronize with and follow the server of application A's Sample service and C's Sample service will synchronize with B's Sample service.
Beware that this can become complex. Consider the following example; in which three machines (FRED, JOE and BOB) each have an instance of applications A, B and C. Application A houses the master service, application B houses the slave service and application C houses the slave of slave service. Assume, also, that the server lists for the service have been configured FRED, then JOE for machines FRED and JOE, but only JOE for machine BOB (i.e. BOB is a client of a client):



Once again, the arrows point to the server instance of each service. Note that, because the slave services cannot be servers, they can only follow their master server location. For application B, the master is housed in application A and so, therefore, all application B slave services will have the current application A server as their current master. Because application B's service instance can not be a server status (rather it is a master to application C's slave service) all application C slaves regard the application B on the same machine as their master.

Be clear about this distinction between server and master: A server instance of a service has an RPCStatus (a pointer to which is returned from the register call) of server(2). A master instance of a service may have an RPCStatus of server(2) OR client(1), depending on whether or not it is server. A slave can not have an RPCStatus of server(2). Any instance can have an RPCStatus of unknown(0).

Revised Code Example

This section will show you how to create a simple cross-application service based on the Sample service created in an earlier section.

First, create a new standard application and copy the sample service code, developed earlier in this document into the new application. Compile this and make sure that it works.
You now have two identical applications, each of which has a different application GUID (look in the SETTINGS.STARTUP file in each application directory to verify this). We will now change the second application so that the "Sample" service works cross application.
Add a RemoteGUID variable to the SampleService declarations.

```
{============================ SampleService
================================}
{ This service maintains a simple synchronizable state and properly
supports  }
{ all entry points required by RPC Manager for correct operation.
}
{=================================================================-
========}
[
  .
  .
  .
  RemoteGUID                    { GUID of the master application
};
  .
  .
  .
]
```

Set the RemoteGUID variable to the application GUID of the first application. Do this before calling \RPCManager\Register. Note that the binary form of the GUID is used. You can obtain the GUID from the first application's SETTINGS.STARTUP file.

```
Init [
  If 1 Main;
  [
    RemoteGUID = GetGUID(1{bin}, "e7a82aa4-21b2-4303-b7d7-1d66353d-
ab35");
    {***** Register the sample service with RPC Manager. *****}
    RPCStatus    = \RPCManager\Register(SvcName, Invalid, Invalid,
                                        Invalid, \RPC_SYNC_MODE);
  ]
]
```

Use the fourth parameter to GetServerChanges. This is only required if you are doing x-app RPC. It is the GUID of the application that is synchronizing to the master service instance. This parameter must be called

RemoteGUID (case insensitive), so that PackRPC and SetDivert will pick it up and work correctly.

```
<
{================== GetServerChanges ==============================}
{ Called by RPC Manager during startup sync, on a server, to get the
package of RPCs which create a synchronizable state on the client
which is in step with the server. }
{=================================================================}
GetServerChanges
(
RevisionInfo { Revision info from GetClientRevision call
made on synchronizing client - unused here };
PackStreamRef { Pointer to var to receive changes };
ClientName { Name of client };
RemoteGUID { The GUID of the app synchronizing to me };
)
```

The Sample service in your second application is now a slave and will remain as a synchronized client to the master Sample service in your first application.

Related Information:

Programming Example: Create a Simple Service

CurSourceAppGUID

The principle of having a service in a second application be a slave to the first application can be extended to as many applications as you wish, i.e. One master service can have slaves in many different applications. It is unusual for the master (or a slave) to need to know which application is making an RPC to it. However, for those occasions when this is needed, RPC Manager provides a variable called RPCManager\CurSourceAppGUID, which contains the binary GUID of the application that sourced the current RPC. Like all the other RPCManager\Cur*** variables, it is valid only for the duration of the RPC subroutine call.

Application Control of Servership

In normal operation, RPCManager controls which service instance is currently the server for a service. The system is simply configured to

instruct RPCManager how its servership control algorithms should operate.
In rare cases, it may be necessary to control the servership of one or more services by the application itself. Consider, for example, a system where maintenance of the system is necessary while the application is running. For operational reasons, the system owners do not want a number of key services to have servership held by the machine undergoing maintenance, but do want the application to be running.
In such circumstances the application needs to either be able to disable a particular machine from owning servership of the key services or, more generally, be able to control which machine is the server for those services. More complex situations could arise where, for example, a specific machine does not want to be considered a candidate for servership.

Related Information:

...RPCManager API

...VTScada Plug-In API

...Service Synchronization

RPCManager API

Rather than adding new algorithms to RPCManager to cope with individual situations, an RPCManager API exists that enables application provided code to control servership. The API consists of three methods:

- ForceServers
- GetInhibitedServiceList
- GetInSyncServers

These API calls are described in the Service Control Methods section.
A higher-level interface still is provided by the VTScada layer. The service ServiceCtrl drives these RPCManager APIs and provides the ability for an OEM or application layer plug-in to be provided which simply determines who should be server.

Related Information:

API Reference

Related Functions:

...ForceServers

...GetInhibitedServiceList

...GetInSyncServers

VTScada Plug-In API

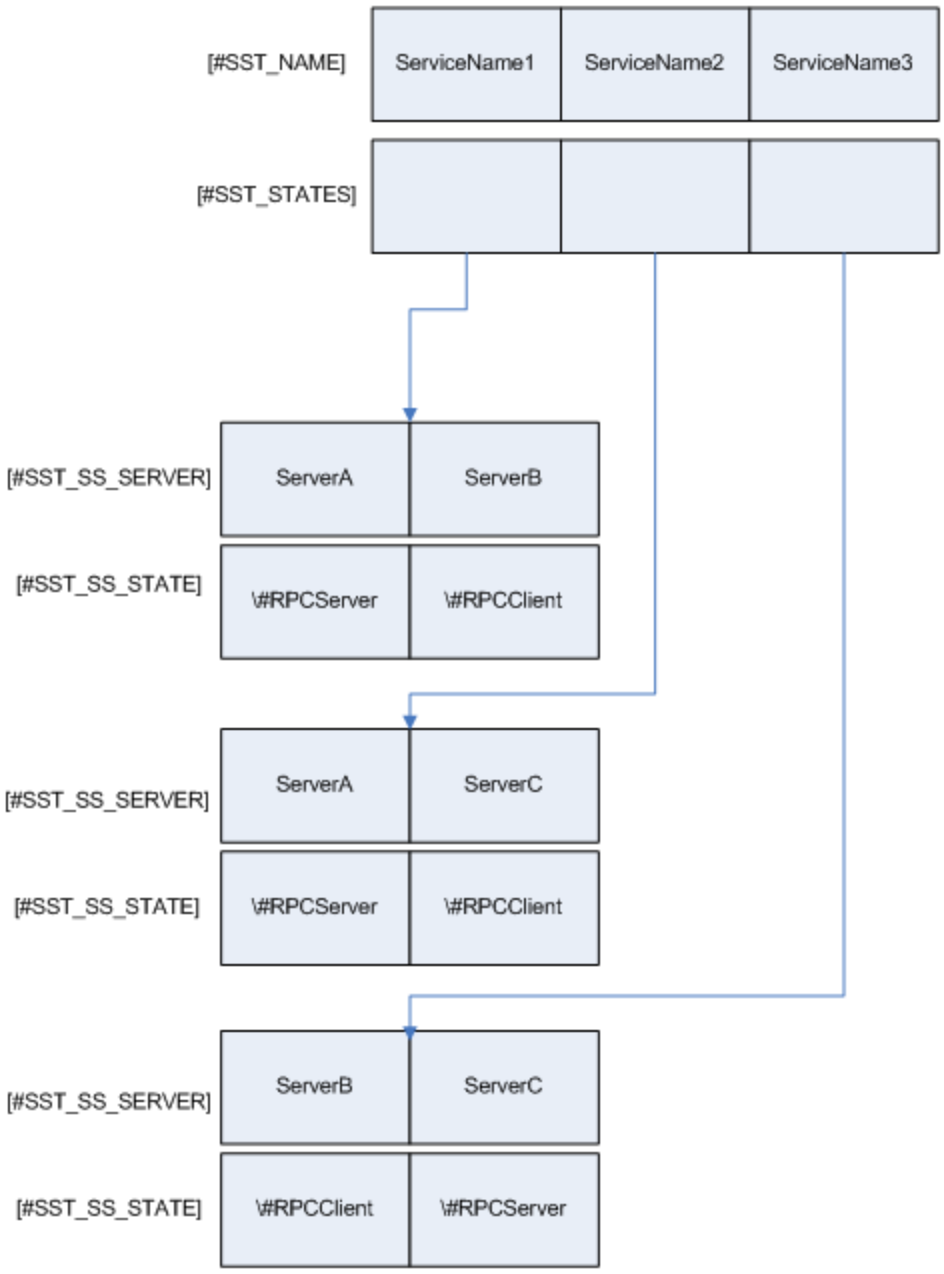
The plug-in must be named ServiceCtrlLogic and is automatically launched (on the application thread) and attached to the ServiceCtrl service when the VTScada application starts.
ServiceCtrl provides an interface that the ServiceCtrlLogic may use. The interface consists of a number of useful variable and constants defining a structure that holds the current server states:

```
  ServerStates                  { Structure holding server states
};
  Trigger = 0                   { Boolean - TRUE on ServerStates
change       };
  RPCStatus                     { Current RPC status of ServiceCtrl
};
  ServiceName = "ServiceCtrl"   { Service name
};
  RPCScopeName = "ServiceCtrl"  { RPC scope name (must match Appmod
decl)     };
  LogicObj                      { Instance of the ServiceCtrlLogic
module     };

  {***** Indices into the ServiceState structure.
        The structure is held as an array, with row #SST_NAME being
the
        service names and row #SST_STATES being arrays of a sub-
structure.
        The sub-structure is an array with row #SST_SS_SERVER being
the
        server name and row #SST_SS_STATE being the server state.
*****}
  Constant #SST_NAME = 0        { Service name
};
  Constant #SST_STATES = 1      { Array of server/states
};
  Constant #SST_NUMELEMENTS = 2 { Keep one bigger than the max above
};
  Constant #SST_SS_SERVER = 0   { Server names
};
  Constant #SST_SS_STATE = 1    { Server states
};
  Constant #SST_SS_NUMELEMENTS = 2 { Keep one bigger than the max
above       };
```

Fundamental to the plug-in logic is an understanding of the ServerStates structure. Essentially, ServerStates is a 2D array with each element of row [#SST_NAME] holding the service names and each element of row [#SST_STATES] holding an array. ServerStates is initialized from the inhibited service list, from RPCManager. There is therefore one column for each service under application control.

Each element of row [#SST_STATES] is also a 2D array, with each element of row [#SST_SS_SERVER] holding the ordered list of servers, highest listed first. Each element of row [#SST_SS_STATE] is initially Invalid. It is the responsibility of the plug-in to maintain this row with the servership state of the service on the corresponding server.

A diagram may help here:



The plug-in must provide an InitializeState subroutine that gets called, in a CriticalSection, when the ServiceCtrl service instance becomes server. It must populate the ServerStates structure's #SST_SS_STATE elements to reflect the servership states for the service. Only one element for each service may be set to \#RPCServer. All others must be set to \#RPCClient. Failure to observe this precaution will result in unpredictable behavior in the services being controlled.

The plug-in is at liberty to change these element states at any time, but has to do so in a CriticalSection, to avoid race conditions with RPCManager. After setting them it must set ServiceCtrl's Trigger variable to 1.
If the plug-in wishes to make an RPC call to other service instances of itself (inside the ServiceCtrl service), it can use the expression Concat (\RPCScopeName, "\LogicObj") as the RPC call scope and \ServiceName as the name of the service in which to make the call.

Service Synchronization

ServiceCtrl is an RPC service. It is therefore subject to normal RPCManager service synchronization rules. To this effect it provides a GetServerChanges method, to be called by RPCManager during service synchronization. This is extended into the ServiceCtrlLogic plug-in via two subroutine methods that the plug-in can provide:

- StartGetServerChanges(). This is called in a CriticalSection at the start of GetServerChanges, allowing the plug-in to pack any RPCs is wishes to have executed during synchronization, before any RPCs that ServiceCtrl may generate as part of its synchronization package. StartGetServerChanges is expected to return a stream of packed RPCs, with the current stream pointer at the end of stream. If no RPCs are packed, StartGetServerChanges must return Invalid.
- EndGetServerChanges(Stream). This is called in the same CriticalSection after ServiceCtrl has packed any RPCs it wishes into the supplied stream. The stream is positioned at end of stream. The plug-in can pack any RPCs it wishes into the stream and is expected to return the stream. If no RPCs are packed into the stream, EndGetServerChanges must return the stream unchanged.

ServiceCtrl will augment the stream with a SetSyncComplete call after EndGetServerChanges has returned.

System Level Services

A system level service is an RPC service whose execution scope lies within the system library (i.e. the root of VTS), rather than within an application. System level services are therefore started when VTScada starts, rather than when an application starts and remain running until VTScada is closed down.
System level services can be used to provide a service that is common to many applications or provide a service that does not rely on the presence of an application.
VTScada defines one such service – the VICManager service. Its function is to coordinate the management of VTScada Internet Clients across a number of computers running VTScada. It must be able to transfer and synchronize VIC server lists and application configurations regardless of whether any of those applications are running. In addition, it provides code that an application can call to obtain state information regarding VIC sessions across all configured computers and to control those VIC sessions.

Related Information:

...Creating a System Level Service – list of RPC-specific exceptions and additions to a standard VTScada service.

Creating a System Level Service

A system level service operates identically to an application service, with the following exceptions and additions:

- It cannot use an application GUID. Instead the system-level GUID must be used. This is an all-zero GUID and is pre-declared in RPCMan-ager\SystemGUID.
- The service must define local variables RemoteGUID and LocalGUID. These must be preset to RPCManager\SystemGUID.
- The service must either provide its own configuration file or use a section in the system SETUP.INI file.

- If the service requires RPC call-backs (see section RPC Call-Backs) it must insert into the array in \ServiceList the object value of the scope in which the call-back method (RPCServerNotice) can be found, for example:

```
{***** Insert myself into the system-level service list *****}
ServiceList = InsertArrayItem(ServiceList, Invalid, Self());
```

Other than the above list, write a system-level service just as you would an application level service.

API Reference

This section contains a reference for each method that RPC Manager provides for external use.
Some methods can be called only as subroutines, whereas some methods can be called either from steady state or as subroutines. These are clearly indicated in each function description. Calling a method that is implemented only as a subroutine from steady state will have undesirable effects. Note that a steady-state call may return Invalid for a brief time before the return value of the method stabilizes.

Related Functions:

...RPC Manager Functions - list with descriptions and links.

...Deprecated RPC Methods

...Server List Source Callback Methods

RPC Manager Functions

BinIP2Text	(RPC Manager Library) Returns a text representation of a specified binary IP in a printable format.
ConnectToMachine	(RPC Manager Library) This subroutine increments the usage count on the specified workstation and forces RPC Manager to attempt to establish a connection with the specified workstation if it is not already connected. Subroutine call only.

DisconnectFromMachine	(RPC Manager Library) This subroutine disconnects from a workstation by decrementing the usage count on the specified workstation and forcing the RPC Manager to attempt to establish a connection with the specified workstation if it is not already connected. Subroutine call only.
ForceServers	(RPC Manager Library) Sets the servership of an application service to a specific state.
GetClientDiverts	(RPC Manager Library) Returns a one-dimensional array of flags, indicating the divert status of each client.
GetClientGUIDs	(RPC Manager Library) Returns a one-dimensional array of the application GUIDs of the clients of the specified RPC service instance.
GetClientIPs	(RPC Manager Library) Returns a one-dimensional array of the IPs of the clients of the specified service instance.
GetClientList	(RPC Manager Library) Returns a one-dimensional array of the names of the clients of the specified service instance. Steady state or subroutine call.
GetClientMode	(RPC Manager Library) Returns a one-dimensional array of the modes of the clients of the specified service instance.
GetClientNodes	(RPC Manager Library) Returns a one-dimensional array of the object values of the MachineNodes of the clients of the specified service instance. Steady state or subroutine call.
GetClientsListed	(Obsolete - RPC Manager Library) Returns a one-dimensional array of the names or IPs of the clients that have been derived from the "-Clients" section of the service configuration file.
GetInhibitedServiceList	(RPC Manager Library) Returns a one-dimensional

	array of the names of all services inhibited from RPCManager servership control.
GetInSyncServers	(RPC Manager Library) Returns a one-dimensional array of the names or IPs of the potential, synchronized servers for the given service.
GetIP	(RPC Manager Library) Returns an IP address for a workstation, given its name.
GetLocalIP	(RPC Manager Library) Returns an IP address for the local workstation that is known to the specified remote workstation.
GetLocalNumber	(RPC Manager Library) Returns the index of the local workstation down the prioritized server list for the named service. Steady state or subroutine call.
GetMachineNode	(RPC Manager Library) Returns the object value of the MachineNode for the specified name or IP. Steady state or subroutine call.
GetMakeAltPtr	(RPC Manager Library) Returns a pointer to a variable containing the Alternate status for the local service instance in the calling application for the specified service. Steady state or subroutine call.
GetRemoteVersion	(RPC Manager Library) Returns the version number of VTScada running on a specified workstation. Steady state or subroutine call.
GetServer	(RPC Manager Library) Returns the name of the active server for a specified service.
GetServerChanges	(RPC Manager Library) Launched by RPC Manager on a service server to obtain the service's synchronization data (i.e. called by RPC Manager during startup synchronization on a server to get the package of RPCs that create a synchronizable state on the client which is in step with the server).
GetServerMode	(RPC Manager Library) Returns the mode in which the

	current server for a specified service is running.
GetServerNumber	(RPC Manager Library) Returns the index down the prioritized server list of the current server for the specified service. Steady state or subroutine call.
GetServerSIDPtr	(RPC Manager Library) Returns a pointer to a variable that holds the session ID for the current server for the specified service.
GetServersListed	(RPC Manager Library) This subroutine returns a one-dimensional array of the names or IPs of the servers that has been derived from the "-Servers" section of the service configuration file.
GetServiceScope	(RPC Manager Library) Returns the service instance for a service.
GetSessionID	(RPC Manager Library) Returns the current session ID for a specified application on a workstation.
GetSocketStatus	(RPC Manager Library) Returns the connection status of either: 1) The machine node if the subnet is not valid, or 2) The socket that is on the specified subnet.
GetStatus	(RPC Manager Library) Returns a variable that holds the current service instance status for the specified service.
IsClient	(RPC Manager Library) Is Client of a Service. This subroutine returns an indication of whether or not a particular workstation is a client connected to a service. Returns 1 for the specified service if the specified machine is a client to the machine on which the IsClient() call is made.
IsMatch	(RPC Manager Library) Determines whether two names or IPs indicate the same workstation. This subroutine returns a "1" if the two names or IPs (any combination) refer to the same workstation.
IsPotentialServer	(RPC Manager Library) Is Potential Server for a Service.

	This subroutine returns an indication of whether or not the local workstation is a potential server for a service. Returns "1" if the local workstation can be a server for the specified service. IsPotentialServer should not be called in steady state.
IsPrimaryServer	(RPC Manager Library) Is Primary Server Active for a Service. This module returns an indication of whether or not the active server for a service is the primary server. Returns "1" if the local workstation is the current server for the specified service.
IsServiceReady	(RPC Manager Library) Is Primary Server Active for a Service. Only available in VTS 6. This module returns an indication of whether or not the specified server is in synchronization with the server instance. Returns "1" if the local instance is in synchronization with the server instance.
PackData	(OBSOLETE) (RPC Manager Library) This method packs an array or set of module instance parameters into a stream.
PackParms	(RPC Manager Library) This method packs supplied parameters into a stream. Subroutine call only.
PackRPC	(RPC Manager Library) Packs an RPC call and a set of parameters into a stream. Subroutine call only.
RecommendAlternate	(RPC Manager Library) Instructs RPC Manager that the local service instance does not consider itself a good server candidate.
RecommendPrimary	(RPC Manager Library) Instructs RPC Manager that the local service instance considers itself a good server candidate.
Register (RPC Manager)	Registers a service for RPC and returns a pointer to the variable containing the current RPC status of the service.

RunPack	(RPC Manager Library) Is unpacks and executes a set of RPCs from a stream constructed with PackRPC.
SetDivert	(RPC Manager Library) Informs RPC Manager that the synchronization state of a service has been sampled during synchronization, and service RPCs for the specified client should be buffered until synchronization completes. Subroutine call only.
SetRemoteValue	(RPC Manager Library) This subroutine sets the specified variable within an application instance on a workstation to the specified value. Subroutine call only.
SetSyncComplete	(RPC Manager Library) Informs RPC Manager that service synchronization is complete as far as the local service instance is concerned. Subroutine call only.
TextIP2Bin	(RPC Manager Library) Returns the Binary representation of the specified IP.
UnpackData	(RPC Manager Library) This method unpacks a stream into an array or set of module instance parameters. Subroutine call only.
UnpackParms	(RPC Manager Library) This method unpacks a stream into the supplied parameters. Subroutine call only.
WriteLock	(RPC Manager Library) This subroutine attempts to require a Write lock for the specified service. Subroutine call only.

Deprecated RPC Methods

The methods in this section are deprecated and have been retained for use by existing, legacy applications. New applications should use \RPCManager\Send() which subsumes all the functionality provided by these legacy methods.
RPC (function call)
SendAll
Broadcast

RPCExecute
RPCExecuteServer
RPCExecuteAll

Server List Source Callback Methods

An object that is passed into the ListSource parameter of \RPCManager\Register() should implement the following methods. Note that usually Invalid is passed into that parameter, in which case the list (if not explicitly provided in the ServerList parameter of the Register call) is retrieved from the application's Servers.XML file. However, providing a ListSource object with the following interface is a way of specifying an alternate source of server lists for the service, should you desire a different way of dynamically updating server lists and settings.

Related Functions:

...ServerListSubscribe

...ServerListUnsubscribe

...GetServerList

...GetRPCServiceSettings

ServerListSubscribe

Description: Called by RPCManager to subscribe to server list changes.

Returns:

Usage:

Function Groups:

Format: ServerListSubscribe(SubscriberObj, Callback)

Parameters:

SubscriberObj

Object that is subscribing to server list changes

Callback

Text name of method in Subscriber to call (with no

parameters) when the server list may have changed

Comments: If the server list cannot be changed dynamically, then this need not be implemented.

ServerListUnsubscribe

Description: Called by RPCManager to unsubscribe from server list changes.

Returns:

Usage:

Function Groups:

Format: ServerListUnsubscribe(SubscriberObj)

Parameters:

SubscriberObj

Object that is unsubscribing from server list changes

Comments: If the server list cannot be changed dynamically, then this need not be implemented

GetServerList

Description: Called by RPCManager (on service registration, and after every subscription callback) to retrieve the server list given a specified server list name. It should return an array of server names/IPs.

Returns:

Usage:

Function Groups:

Format: GetServerList(ServerListName)

Parameters:

ServerListName

Name of server list to retrieve

Comments:

GetRPCServiceSettings

Description:

Returns:

Usage:

Function Groups:

Format: GetRPCServiceSettings(ServerListName)

Parameters:

ServerListName

Name of server list whose settings we are retrieving

Comments: It should return a dictionary of service settings, typically a dictionary with a single "Sticky" element, whose value is the sticky flag for the specified service.

Diagnostics

Earlier versions of the RPC Manager had an inbuilt diagnostic facility, which could be activated by setting the system variable \RPCManager\RPCDiagnostics to a non-zero value. Setting this back to zero will disable the diagnostics again.
This application has been superseded by the Trace Viewer application.

Related Information:

...Trace Viewer Application.

RPC Routing and Execution

Related Information:

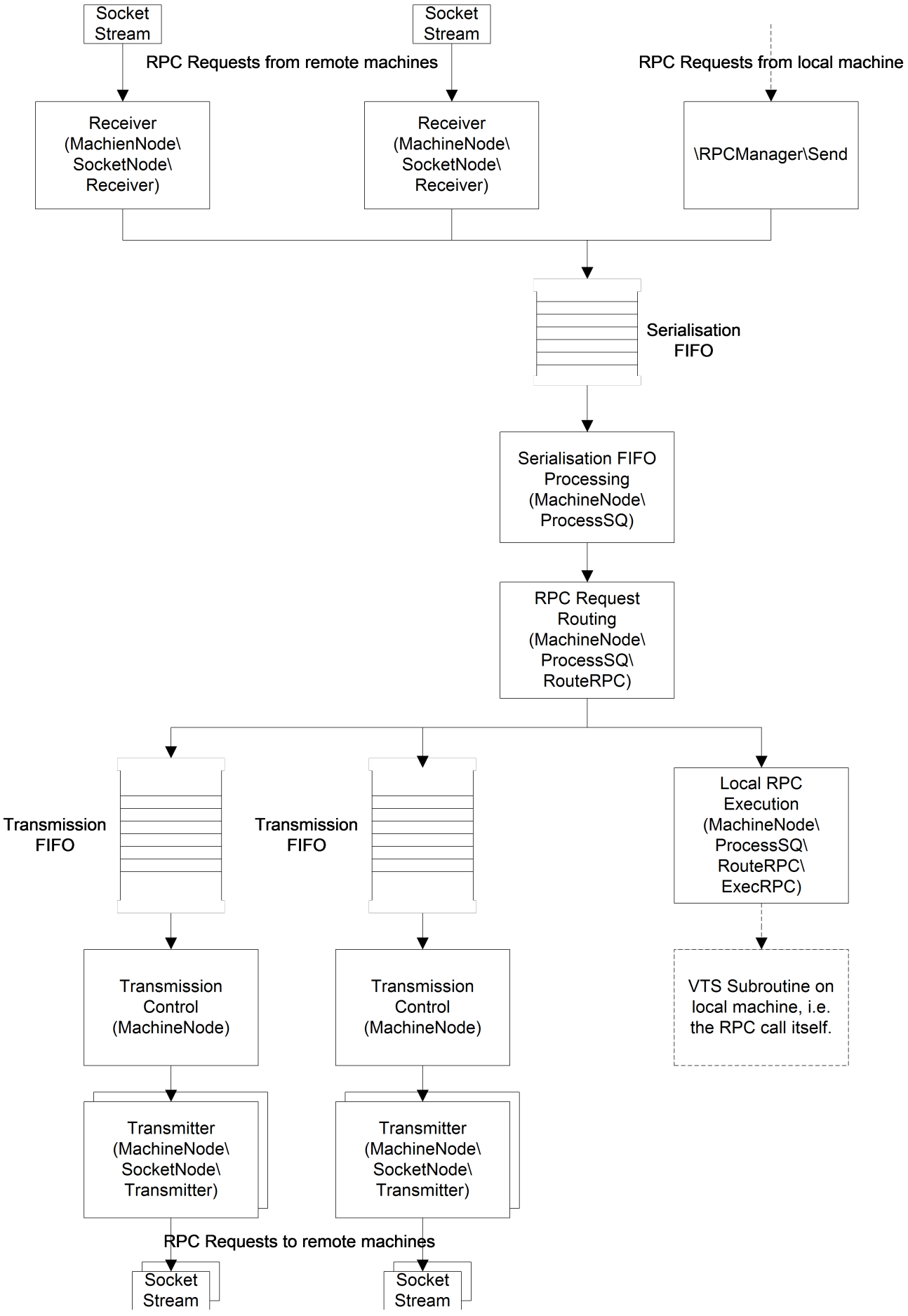
RPC Internal Routing

RPC Manager absolutely guarantees that a sequence of RPCs generated on one machine, will be routed to the next consumer of the request in the order that the RPCs were generated.
What this means to the programmer is that two consecutive RPCs that are generated on the same machine as each other and are to execute on the same target machine will always be executed in the order in which they were generated. This assumption promotes robust algorithms when programming for a distributed system. This assumption is necessary for the correct operation of services and other parts of the RPC subsystem.
In the interests of preventing one large RPC blocking others, the version 4 RPCManager only preserves this rule within a service or between directed RPCs. An RPC issued from service A is not guaranteed to execute before an RPC from service B, even if it was issued first. Two RPCs within service A are, however, guaranteed to execute in the order they were issued.
To achieve this, RPC Manager serializes all RPC requests through a single FIFO queue on the local machine. The serialization includes not only locally generated requests, but also requests that arrive from other machines.
RPC requests are then removed, one at a time, from the serialization FIFO and examined to determine:

- If the request should be forwarded to other RPC manager instances in the distributed system. There the request will be processed in an identical fashion.
- If the request should be executed on the local machine.

In versions of RPCManager prior to version 4, each MachineNode contains a transmission FIFO, ensuring that RPC requests are delivered to the remote machine in the strict order that they were generated on the local machine.

From version 4 onwards, each MachineNode contains one FIFO for directed RPCs and one FIFO for each service that is registered with RPCManager.
MachineNode selects the SocketNode to encode and transmit the request, according to the methods outlined in the section on Multi-homed Systems.
Each SocketNode\Receiver module accepts and decodes the incoming requests and places them on the local serialization FIFO for processing.
The flow of RPC requests through the RPC subsystem is depicted in the following diagram, where the parenthesized names are the RPC Manager module that performs the operation:


Socket Stream
Socket Stream
RPC Requests from remote machines
RPC Requests from local machine
Receiver (MachienNode\ SocketNode\ Receiver)
Receiver (MachineNode\ SocketNode\ Receiver)
\RPCManager\Send
Serialisation FIFO
Serialisation FIFO Processing (MachineNode\ ProcessSQ)
RPC Request Routing (MachineNode\ ProcessSQ\ RouteRPC)
Transmission FIFO
Transmission FIFO
Local RPC Execution (MachineNode\ ProcessSQ\ RouteRPC\ ExecRPC)
Transmission Control (MachineNode)
Transmission Control (MachineNode)
VTS Subroutine on local machine, i.e. the RPC call itself.
Transmitter (MachineNode\ SocketNode\ Transmitter)
Transmitter (MachineNode\ SocketNode\ Transmitter)
RPC Requests to remote machines
Socket Stream
Socket Stream

The settings that were provided by the original \RPCManager\Send() sub-routine call provide the initial routing information. RPC Manager may modify this information before queuing the RPC request on a transmission FIFO, so that the receiving RPC Manager will make the correct decisions about any further routing. Further routing only occurs in the "client of a client" case (see the section Clients of Clients).
To achieve this, the RPC request carries a set of "routing flags" with it, which are transported across machines. These flags and the routing strategy are discussed in section RPC External Routing.
With the Diagnostics window's "Detail Trace" button pressed, the routing of RPC requests between the serialization FIFO and the transmission FIFOs are revealed. A request received from a remote machine is diagnostically recorded when it is posted to the serialization FIFO, when it is removed from the serialization FIFO for processing and when it is queued on each transmission FIFO.
Without this button pressed, an RPC from a remote machine will be recorded only when it is removed from the serialization FIFO or when it is queued on each transmission FIFO.

RPC External Routing

All RPC requests carry a set of routing flags which determine how the RPC Manager which is processing the request, will route the message and whether it will execute the RPC locally.
The flags are bit-significant:

Flag Name	Bit
#FNR_EXECUTE	20
#FNR_IFSERVER	21
#FNR_TOCLIENTS	22
#FNR_RECURSIVE	23
#FNR_EXSERVERS	24

The flags are initially generated when the \RPCManager\Send() call is made:

1. If the RPC is a directed RPC, or the ExecLocally parameter is greater than zero, #FNR_EXECUTE is set.
2. If the SendServer parameter is greater than zero, the #FNR_IFSERVER and #FNR_EXECUTE flags are set.
3. If the SendAllClients parameter is greater then zero, the #FNR_TOCLIENTS flag is set.
4. If the Recursive parameter is greater than zero, the #FNR_RECURSIVE flag is set.

When the RPC request is removed from the serialization queue and processed, the following rules are applied, in the order shown to implement the routing algorithm:

1. If the RPC is a directed RPC:
    a. If the RPC is to be run on the local machine, execute it.
    b. Otherwise, place it on the transmission FIFO for the target machine.
2. If the RPC is a service RPC:
    a. If the RPC has been received from a remote machine and has the #FNR_IFSERVER set, but the recursive flag is clear, the RPC request will be inhibited from propagating to a higher-order server. This is a somewhat special case, where an RPC request from a client-of-a-client is not to be continually propagated to subsequent servers. To achieve this, the #FNR_IFSERVER flag is forcibly cleared, so the following rules can continue to be applied.
    b. If the #FNR_IFSERVER flag is still set and the local machine is not the service server (from the point of view of the local machine) and the service filtering mode permits the request to be transmitted, it is queued on the transmission FIFO for the service server, with the same flag settings as the request.
    c. If the #FNR_IFSERVER flag is still set and the local machine is the service server and the #FNR_TOCLIENTS flag is set, the RPC is queued on the transmission FIFOs for each client of this server. If the #FNR_EXSERVERS flag is set, the RPC is only queued if the client is a potential server from the point of view of the local machine. If the #FNR_

RECURSIVE flag is set, the request is queued with the #FNR_EXECUTE, #FNR_TOCLIENTS and #FNR_RECURSIVE flags set to cause it to be executed on the remote client and propagated to clients the client. If the #FNR_EXSERVERS flag is set, this is propagated to the potential servers as well. If the #FNR_RECURSIVE flag is clear, only #FNR_EXECUTE is set on the outgoing request.

d. If the #FNR_IFSERVER is clear and the #FNR_TOCLIENTS flag is set, the RPC is queued on the transmission FIFOs for each client of this machine, for the service. The flag settings used in c, above, are used for propagation of requests to other machines. This rule implements the transmission to a client of a client.

The above information is of value when identifying why an RPC that you expected to go to a particular machine was not transmitted to it.

RPC Execution

An RPC request is executed on the local machine whenever:

1. The RPC request is a directed RPC, aimed at this machine.
2. The RPC is a service request and the #FNR_EXECUTE flag is set, unless the #FNR_IFSERVER flag is also set and this machine is not the service server (from the point of view of this machine).

The values of \RPCManager\CurSocketNode and \RPCManager\CurSessionID are set to reflect the machine that the RPC request was received from. If the RPC was sourced from the local machine, these will have valid values appropriate for the local machine.

RPC Security

VTScada security is application based. Usernames and passwords are held by an application and are used to authorize the actions that a user can perform. RPC security is system based and is concerned with ensuring that RPC communication between VTScada servers is secure. Inter-server communication security can be sub-divided into two parts:

- Security of data. This pertains to the protection against modification or discovery of system data.
- Security of system. This pertains to the protection of the system against malicious network traffic and unauthorized hands-on modification of the system or the plant it controls.

Research shows the latter to be the area of almost all reported vulnerabilities of SCADA systems. The vulnerabilities cause loss of service, a crash or other catastrophic failure of the SCADA system. Discovery of these vulnerabilities is usually made in an environment other than a production environment (i.e. someone has a copy of the software and sets out to break it in a lab environment) and, therefore, physical access to the system is not implemented.

Related Information:

...Security Measures

...[RPCManager-AllowIP]

Security Measures

Best practices ensure best security and a multi-tiered approach is recommended:

- Physical access to the servers comprising a SCADA system needs to be restricted to only those who need access.
- Networks also require physical security. If it is not possible to connect a computer to the SCADA system servers, the attack surface is significantly reduced. This can be achieved either by preventing physical access to the networking infrastructure or by using a router to provide isolation of the network.
- Network access to the servers. In addition to the protection afforded by limiting access to the network on which the servers are communicating, protocol security can be added to afford additional protection against compromise of the network. Server class Windows operating systems are quite capable of automatically securing communication to a specific machine using IPSec (and requiring that all other servers and workstations comply) by configuring IP Security Policies using the MMC IP Security Policy

Management snap-in and either using Windows X.509 Certificate Management or purchasing third-party trusted certificates. This affords access restriction to nominated computers only (those with appropriate certificates installed) and prevents data snooping and tampering by using encryption and message digests. For a wide area network, secure tunnels must be used, e.g. an IPSec VPN connection between sites.

[RPCManager-AllowIP]

While preventing network access using IPSec is recommended, smaller systems running in trusted environments may wish to simply prevent inadvertent connection to a live VTScada system by, say, a development or test system.
The [RPCManager-AllowIP] section (in SETUP.INI) is used to achieve this. This can also be used to provide yet another tier in the security model. This section is not present by default. Its inclusion causes RPCManager to refuse connections from all external computers and to refuse to establish socket connections to any external computer.
Computers that should be allowed to connect via VTScada RPC are then added to the [RPCManager-AllowIP] section. Modification to this section do not take effect until VTScada is restarted.
Each entry consists of a single line specifying the IP address of a peer with which connections are permitted. Names are not acceptable, nor are ranges of IPs. This is rarely a limitation, as the number of server systems tend to be small and their IPs are normally static (recommended).
An example section might look like this:

```
[RPCManager-AllowIP]
  IP = 192.168.1.5
  IP = 192.168.1.6
  IP = 192.168.1.7
```

This will allow VTScada RPC connections from only the three listed IP addresses. It is harmless to specify the IP address of the local computer in this section. This eases installation and maintenance of a system with, for example, all three servers in a system being listed in the SETUP.INI file and the same file installed on each server.

Configuration

There are two types of initialization data used by RPC Manager:

- Configuration used to control the behavior of the distributed system as a whole or a machine within the distributed system.
- Configuration used to control a service.

Non-service configuration is held in the SETUP.INI file and has been designed to allow the specification of the non-service configuration for an entire distributed system identically on all machines within the distributed domain.

Service configuration, on the other hand, is specified in the application layer's Servers.XML file, if the \RPCManager\Register() call's ListSource parameter is Invalid. Otherwise, it depends on the ListSource's API functions (see Server List Source Callback Methods section).

The server list from a layer's Servers.XML that a service uses is based on the following order of precedence (given that the server list name is either the ListName parameter to the service's Register call, or, if that isn't provided, then the ServiceName in the Register call):

1. A server list specific to the current workstation and server list name.
2. A server list specific to the current server list name, but not workstation-specific.
3. A server list specific to the current workstation, but not service-specific.
4. The default server list.
5. If none of these lists are present in Servers.XML, then this machine is treated like the solitary server.

Related Information:

...SETUP.INI [System] Values for RPC - list of control settings.

...Variables available in \RPCManager - publicly exposed settings.

...Application Settings for RPC - control of application-level behavior including driver setup and alarm displays.

...Name Resolution - methods of discovering the IP address of a machine.

SETUP.INI [System] Values for RPC

The following variables can be set in the [System] section of SETUP.INI, located in the VTScada installation directory.

Related Information:

...Setup.ini [RPCManager-ExcludeIP] - (See: VTScada Developer's Guide) designation of excluded addresses. See also: RAS Clients

...Setup.ini [RPCMANAGER-NETPRIORITY] - (See: VTScada Developer's Guide) configuration of multi-homed systems. See also: Multi-homed Systems

...Setup.ini [LINKTOLERANCE] - (See: VTScada Developer's Guide) see also, the discussion: Link Tolerances

...[RPCManager-AllowIP] - designation of allowed addresses. See also: RPC Security

...RPCBufferLength

...RPCConnectPort

...RPCDiagnostics

...RPCMaxPacketSize

...RPCMaxQLen

...RPCMaxStartDelay

...RPCMemBuffLimit

...RPCMemSendLimit

...RPCPingInterval

...RPCReconnectTime

...RPCResendDelay

...RPCServerPort

...RPCSktConnectAttemptMax

...RPCSktResendAttempts

...RPCSocketDeadTime

...RPCSocketResendAttempts

...RPCTrace

...RPCUseBuffered

RPCBufferLength

This is the maximum TCP/IP buffer length to use for RX and TX buffers. Prior to version 4, the default is 33,554,432 bytes if not defined. This has no bearing on the size of the RPC packets transmitted and should be left to default settings.
From version 4 onwards, the default is 262144. This defines the maximum amount of data from the socket connection that Windows will use. If VTScada does not drain the buffers, further transmissions from the remote end of the socket connection will cease until it is drained. This also defines the TCP "window size" that VTScada uses.
There is an important relationship between this value and RPCMaxPacketSize. RPCManager reads the packet header from the socket and then waits for sufficient data to become available before reading it. This is intentional and reduces the amount of buffer that VTScada must allocate and the amount of script code processing that needs to be done until the entire packet has been received. This effectively reduces the "attack surface" of RPCManager by limiting the amount of data that can be maliciously pushed into a socket connection.
Therefore, for version 4 onwards, the transmitting machine's RPCBufferLength MUST be greater than RPCMaxPacketSize by at least 10%. Otherwise, you will end up with a zero-length TCP window and lose your connection.
The default values will work for LANs and WANs.

Related Information:
RPCMaxPacketSize

RPCConnectPort

The TCP/IP port that RPC Manager tries to connect to.
Defaults to the value of RPCServerPort.

Related Information:
RPCServerPort

RPCConnectStrategy

The IP addresses that the RPC Manager can open in order to connect to remote machines , can be provided by two sources:

- Name resolution.
- A set of IP addresses, supplied by the other machine when the VTScada RPCManager connects to it.

The value of RPCConnectStrategy controls how these are used, as follows:

Value	Connect Strategy
Invalid	If a DNS server is being used for name resolution, opens each IP from the DNS query in the order supplied by DNS. If no DNS server, opens each IP supplied by the remote machine. This is the default setting for all VTScada versions after 10.2.06.
FALSE	Opens each IP supplied by name resolution, regardless of whether a DNS server is being used or not and ignores the IPs supplied by the remote machine.
TRUE	Initially opens only the first IP address supplied by name resolution and then opens each IP supplied by the remote machine.

RPCDiagnostics

Set to a non-zero value to display the RPC Diagnostics window on startup. This value may also be set at run time by scoping into the system and changing the value.

RPCMaxPacketSize

Specifies the maximum size of an encoded remote procedure call before the call will be fragmented over a number of transmissions.
From version 4 onwards, specifies the maximum size of an RPC packet. RPCs will be packed into a packet until there are no more to send, or RPCMaxPacketSize is reached. The default is 65536 bytes.

For version 4 onwards, the value of this configuration value must be less than RPCBufferLength by at least 10%. See the discussion of RPCBuffer-Length for more detail.

RPCMaxQLen

Specifies the maximum number of RPC messages destined for remote machines that will be queued before the queue is deemed to have "flooded" and the session closed with all messages lost. The local RPC queue is not subject to this limit. The larger this limit, the more tolerant the system will be to large bursts of RPC activity. However, the larger the queue, the more RAM will be required when the queues are large. This parameter cannot compensate for the condition where the average RPC traffic load exceeds the bandwidth of the network connection.
Default: RPCMaxQLen = 65536

RPCMaxStartDelay

Controls startup behavior if the workstation name is not valid. RPC Manager will wait for a maximum of the seconds specified in RPCMaxStartDelay before assuming there is no network available.
This is useful for auto starting a machine after a reboot, since it is possible for network services to not yet be started when VTScada is starting. RPC Manager will wait indefinitely for the network to start, if this value is set less than 0.
Default: RPCMaxStartDelay = 30

RPCMemBuffLimit

Specifies the number of bytes on a received RPC message that will be kept in RAM before the message is transferred to a temporary file to conserve memory. The larger this value, the faster larger messages will be processed, but correspondingly more RAM will be required.
Default: RPCMemBuffLimit = 2097152
(2Mb)

RPCMemSendLimit

Specifies the number of bytes on a transmitted RPC message that will be kept in RAM before the message is transferred to a temporary file to conserve memory. . The larger this value, the faster larger messages will be processed, but correspondingly more RAM will be required.
Default:RPCMemSendLimit = 1048576
(1Mb).

RPCPingInterval

Specifies the time that a socket can have no data transmitted before a "ping' packet will be sent to all the receiving end to determine that the socket it still good. Will use no value less than 5. The RPCPingInterval is subject to the ToleranceFactor on the receiving machine. The receiver will wait 3 times this long or RPCReconnectTime, whichever is longer, to disconnect if no data is received.
Default: RPCPingInterval = 5

Related Information:
Setup.ini [LINKTOLERANCE] (See: VTScada Developer's Guide)

RPCReconnectTime

Specifies the seconds to wait for data on a socket before disconnecting the socket. The actual delay before disconnect is the greater of this time and three times the RPCPingInterval. RPCReconnectTime is NOT subject to the ToleranceFactor.
Default: RPCReconnectTime = 15

Related Information:
Setup.ini [LINKTOLERANCE] (See: VTScada Developer's Guide)

RPCResendDelay

Specifies the time to wait for an acknowledgment after sending a packet before resending the packet. RPCResendDelay is subject to the ToleranceFactor on the receiving machine.

Default (and minimum): RPCResendDelay = 3

Related Information:

...Setup.ini [LINKTOLERANCE] (See: VTScada Developer's Guide)

RPCServerPort

The TCP/IP port that RPC Manager "listens" on.
Default: RPCServerPort = 5780

Related Information:
RPCConnectPort

RPCSktConnectAttemptMax

Specifies the number of attempts to open a socket before it is declared to be closed. The time between starts at 5 seconds and grows by 10% after each attempt. If a socket is closed and there is no backup socket for the session, then the session is lost and service synchronization must occur when the socket is eventually opened and a new session established. The minimum value is 1.
Default: RPCSktConnectAttemptMax = 5

RPCSktResendAttempts

There will be a maximum of RPCSktResendAttempts to transmit the RPC successfully. If, after these retries, the RPC has still not been acknowledged, the socket stream is terminated and SocketNode goes through its link re-establishment cycle.

Default RPCSktResendAttempts = 5

RPCSocketDeadTime

Specifies the number of seconds that a session will remain alive with no socket connection. This time is AFTER the RPCSktConnectAttemptMax has expired.
Default (and minimum) RPCSocketDeadTime = 1

RPCSocketResendAttempts

Specifies the number of packet re-sends that will occur without acknowledgment, before the socket is closed.
Default (and minimum) RPCSocketResendAttempts = 5

RPCTrace

Set to a non-zero value to log all RPC activity to the disk file "RPCTRACE.TXT", in the VTScada installation directory. This variable only has effect while RPCDiagnostics is set non-zero.
Default: RPCTrace = 0

RPCUseBuffered

When non-zero this will cause RPC to read the TCP/IP IP and Name once only from the low-level socket stream and cache this information. This value should only be set to 0 if the TCP/IP IP or Name can change dynamically. This is not recommended.
Default: RPCUseBuffered = 1

Variables available in \RPCManager

The following variables are available within RPCManager for public consumption, but are strictly read-only. Modifying any of these variables could lead to unexpected behavior.

Started

Initially zero, this gets set to a non-zero value once the RPC Manager has initialized. You should not attempt to register your service until RPC Manager has set this value.

WkStnIP

The IP address of the local workstation. Only valid once Started is set non-zero.

WkStnName

The name of the local workstation. Only valid once Started is set non-zero.

WkStnVersion
The VTScada version running on the local workstation. Only valid once Started is set non-zero.

WkStnIPList
An array, each element holding a textual representation of the set of IPs this workstation is known by.

Related Information:

...Application Settings for RPC

Application Settings for RPC

The following list of variables pertain to remote procedure calls and the RPC Manager.

Related Information:

...ABSharedRPC

... CIPENIPSharedRPC

... DataradioSharedRPC

... DDESharedRPC

... DNP3SharedRPC

... DriverSetupDelay

... MDSSharedRPC

... ModiconPortSharedRPC

... ModiconSharedRPC

... OmronSharedRPC

... OPCClientSharedRPC

... RemCfgTransLog

... SiemensS7PortSharedRPC

... SiemensS7SharedRPC

ABSharedRPC

Indicates whether the same RPC service should be used for all instances of the Allen-Bradley tag type.
If set to 1 (true), then the same RPC service will be used for all instances of the Allen-Bradley tag type.
Default: ABSharedRPC = 0
Section: System

CIPENIPSharedRPC

Indicates whether or not the same RPC service will be used for all instances of the CIPENIP driver.
If set to 1 (true), the same RPC service is used for all instances of CIPENIP.
Section: System
Default: CIPENIPSharedRPC = 0

DataradioSharedRPC

Indicates whether or not the same RPC service will be used for all instances of Dataradio.
If set to 0 (false), the same RPC is NOT be used for all instances of Datar-adio (default).
Section: System
Default: DataradioSharedRPC = 0

DDESharedRPC

Indicates whether or not the same RPC service will be used for all instances of DDE.
If set to 0 (false), the same RPC is NOT be used for all instances of DDE (default).
Section: System
Default: DDESharedRPC = 0

DNP3SharedRPC

Indicates whether or not the same RPC service will be used for all instances of DNP3.
If set to 0 (false), the same RPC is NOT be used for all instances of DNP3 (default).
Section: System
Default: DNP3SharedRPC = 0

DriverSetupDelay

Indicates the number of seconds a VTScada driver waits before trying to resend data once an attempt has failed.
Section: System
Default: DriverSetupDelay = 60

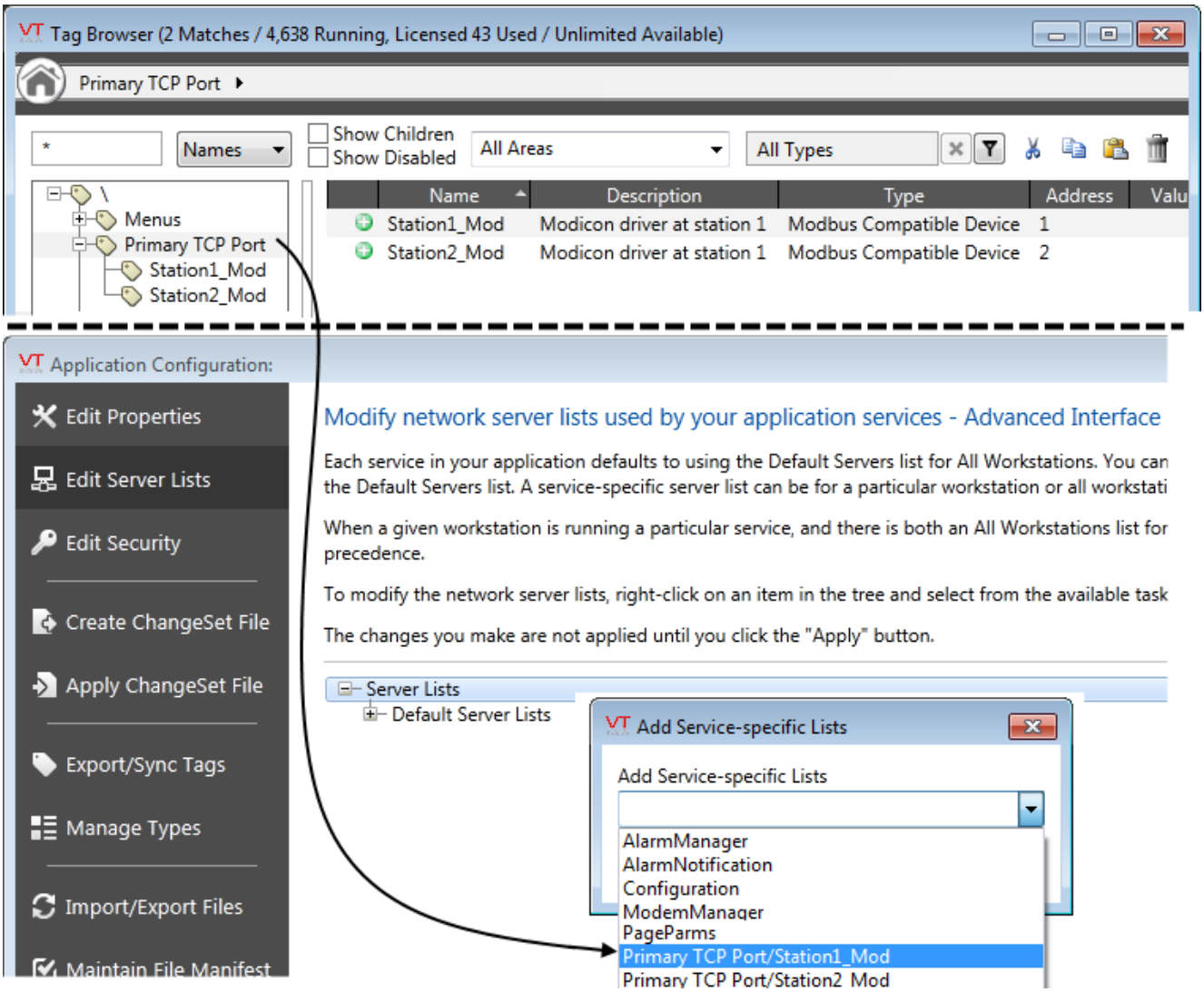
MDSSharedRPC

Indicates whether or not the same RPC service will be used for all instances of MDS.
If set to 1 (true), the same RPC service is used for all instances of MDS.
Section: System
Default: MDSSharedRPC = 0

ModiconPortSharedRPC

Controls whether or not the same RPC service should be used for all instances of the Modbus Plus tag type that are connected to the same serial port or TCP/IP connection.
ModiconPortSharedRPC enables Modbus devices that share the same serial port or TCP/IP connection to be grouped with the same device, enabling Modbus I/O that uses different radio channels to be polled from separate PCs.
If set to 0 (false), then the same RPC service is not used for all instances of the Modicon tag type that are connected to the same serial port or TCP/IP connection (default).
Section: System

Default: ModiconPortSharedRPC = 0
Related Variables: the behavior of the ModiconSharedRPC will be overridden when this property is equal to 1 (true).
Note for multi-server applications using advanced server lists: If ModiconPortSharedRPC is set to 1, each Modbus-compatible driver service will be renamed to a combination of "ModiconServer" followed by the port name.
For example, if the Modbus Plus tags are attached to a driver named "PrimaryTCPPort" in an application where ModiconPortSharedRPC has been set to 1, then the driver service will be named "ModiconServerPrimaryTCPPort".



ModiconSharedRPC

Controls whether or not the same RPC service should be used for all instances of the Modicon tag type.
If your networked application uses a polling driver, then it is recommended that this variable be set to 1.
If set to 0 (false), then the same RPC service is not used for all instances of the Modicon tag type (default).
Section: System
Default: ModiconSharedRPC = 0
Related Variables: This property will be overridden when ModiconPortSharedRPC property is set TRUE (1).

OmronSharedRPC

Indicates whether or not the same RPC service should be used for all instances of the Omron tag type.
If set to 0 (false), then the same RPC service is not used for all instances of the Omron tag type (default).
Section: System
Default: OmronSharedRPC = 0

OPCClientSharedRPC

Indicates whether or not the same RPC service should be used for all instances of the OPC Client Driver tag type.
If set to 0 (false), then the same RPC service is not used for all instances of the OPC Client Driver tag type (default).
Section: System
Default: OPCClientSharedRPC = 0

RemCfgTransLog

Indicates whether or not configuration database transactions should be logged.
If set to 0 (false), then remote configuration database transactions are not logged.

Section: System
Default: RemCfgTransLog = 0

SiemensS7PortSharedRPC

Indicates whether or not the same RPC service will be used for all instances of the SiemensS7 tag type connected to a common TCPIP/Serial port If set to 0 (false), the same RPC is not be used for all instances of SiemensS7Port (default).
If set to 1 (true), the same RPC service is used for all instances of SiemensS7Port.
Section: System
Default: SiemensS7PortSharedRPC = 0

SiemensS7SharedRPC

Indicates whether or not the same RPC service will be used for all instances of the SiemensS7 driver.
If set to 1 (true), the same RPC service is used for all instances of Siemens.
Section: System
Default: SiemensS7SharedRPC = 0

Name Resolution

RPC Manager uses name resolution to discover the IP for a particular machine name. The correct configuration of name resolution services is outside the scope of this document. Reference should be made to the operating system documentation.

Note: Caution: A correctly configured name resolution system is essential for correct operation of the distributed system.

The default name resolution setup for Windows TCP/IP networks looks in the following places, in the following order, in order to derive an IP address:

1. The operating system HOSTS file is searched.
2. If not found, any configured domain name servers (DNS) are checked (including WINS servers).
3. If still not found, NetBIOS broadcasting is used.

If you do not configure any name resolution system, the default behavior of Windows will cause NetBIOS broadcasts to be used. While this will work fine on most network configurations, it may not work properly on large, segmented networks.

The basic rule is that if you can reliable ping a machine by name and have the correct IP addressed by the ping packets, then VTScada will work correctly.

Most SCADA systems will function well with static IP to name mapping. This can be achieved through HOSTS files on individual machines, or by a centralized name resolution service, such as WINS.

VTScada will, however, also function correctly with dynamic IP address assignment, typically provided by DHCP. RAS clients also often use dynamic IP assignment when connected into a RAS server.

If NetBIOS broadcasting is insufficient to meet your name resolution needs, then you will need to consider one of the following alternatives:

Techniques for providing name resolution:

...HOSTS File

...Centralized Name Resolution

...RAS Clients

...Fully Qualified Domain Names

HOSTS File

The simplest approach is to configure up a master HOSTS file and ensure that every machine in the distributed system has the same HOSTS file. For example:

```
127.0.0.1    localhost
192.168.3.21 Server1
192.168.3.22 Server2
192.168.3.23 Server3
192.168.3.30 OpRoom1
192.168.3.31 OpSuper
```

```
192.168.3.50 RemoteWorks
192.168.3.55 RemoteWorks1
```

The disadvantage of this method is that each machine in the distributed system must have a consistent set of information.

Centralized Name Resolution

In environments where new machines may be added on a regular basis, maintaining a consistent set of HOSTS files may be an unacceptable overhead. In such cases, a centralized name resolution system is preferable. This is normally provided by one or more designated systems, often domain controllers and is often termed DNS. The assignment of IPs to machine network interfaces can still be performed statically, or can be dynamically assigned.
Dynamic IP assignment refers to the ability of a designated system to dynamically assign an IP address from a pool of IP addresses, for a machine connected to it.
VTScada provides support for dynamic IP assignment.
For robustness, SCADA systems are generally configured with static IP address assignments, reserving dynamic IP assignment for multiple dial-in links.

RAS Clients

A Remote Access Service (RAS) server provides access to remote computers. Generally, the RAS server is configured to provide a separate IP address for itself and the remote client, when connected. Allocating each client a different IP address from a pool of addresses caters for multiple concurrent clients.
VTScada provides support for multiple remote clients.
When allowing a remote client to connect, due consideration should be given to deciding whether the remote client will require access to the LAN that the RAS server is connected to.
If access is required, then it is better to delegate the RAS server to be a machine other than one running VTScada. In this way:

1. Routing between the RAS server and the SCADA system is handled by the network infrastructure.
2. The RAS server can be shared between infrequent access to the SCADA system and other work, without compromising the SCADA system.
3. "Hacking" attacks, e.g. denial-of-service (DoS), are less likely to disable your SCADA system, when the point of access is separated from the SCADA system.

If access is not required, or another system is not available to be a RAS server, then you can use a machine running VTScada as the RAS server.

Note: Caution: If the RAS server is also running VTScada, prior to version 5.1502, then, for correct operation, it is essential that the RAS IP addresses appear on a different subnet to any Network Interface Cards (NIC). If this precaution is not observed, the attached RAS client will be able to access the NIC IP addresses on the same subnet. This will not compromise operation, but will severely impair RPC Manager's performance.

From VTS version 5.1502 onwards, a machine running VTScada can accommodate RAS clients on any subnet, including one already used by a LAN connection. Instructing RPC Manager, via a SETUP.INI section, to not create a connection to specific IP addresses, achieves this. By specifying the IP that the RAS host presents as its own IP to the RAS client, the RAS client will not create a connection to the RAS host IP, but only connections to the other IPs that the host machine is known by.
For example, if a machine running VTScada had an IP of 192.168.0.40 and that machine was configured to support a RAS client, such that the RAS client would see the host machine as 192.168.0.150 and the RAS client be assigned an IP address from a pool of addresses from the range 192.168.0.151 to 192.168.0.155, then the following section should appear in the RAS client's SETUP.INI file, so that the RAS client will only make a connection to 192.168.0.40 [which will be done over the RAS link] and not to 192.168.0.150:

```
[RPCManager-ExcludeIP]
  IP = 192.168.0.150
```

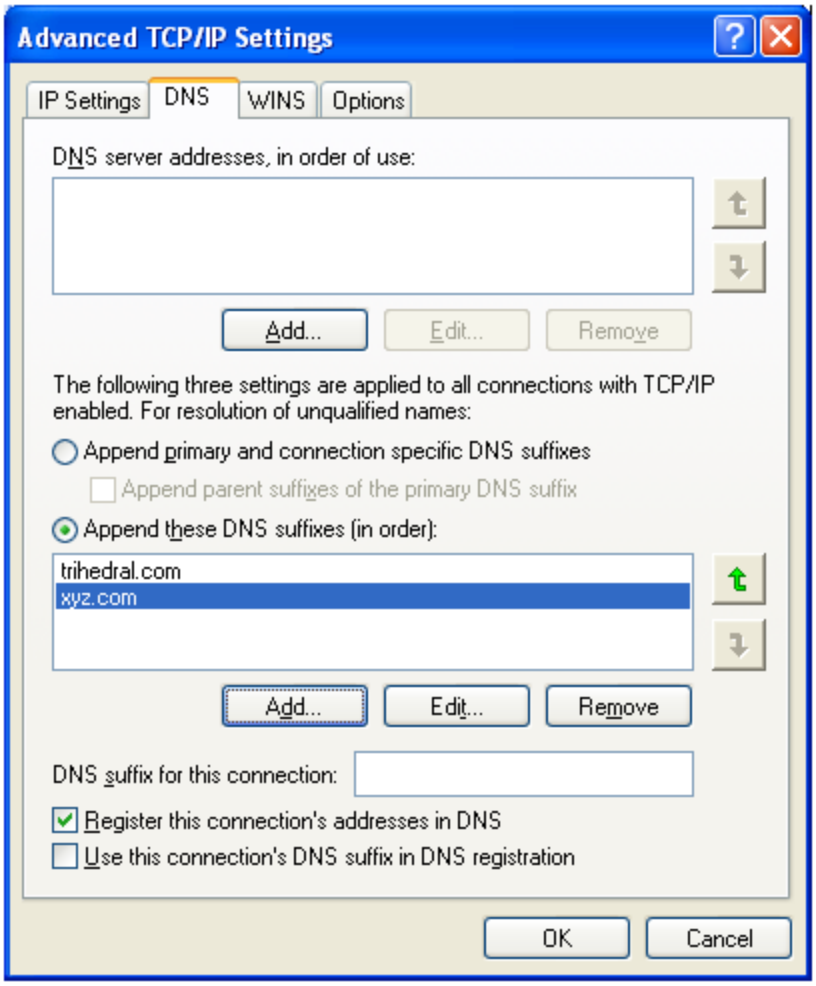
Note that this is not necessary if the RAS IP address pool is on a different subnet from any other IP of the RAS host, so long as no routing exists between the two subnets.
From VTS version 5.18 onwards it is not necessary to exclude any IP addresses on the server, however, if the IP cannot be accessed by a connecting client VTScada system, it is advisable to exclude it from RPC Manager's view by the above method.

Fully Qualified Domain Names

A fully qualified domain name [FQDN] takes the form <host>.<domain>, e.g. xx.xyz.com. When viewing the SocketNodes, displayed by RPC Diagnostics, you may notice that the MachineNode names are machine names, whereas the SocketNode names are FQDNs. This is because SocketNode names are obtained by reverse name lookup, converting the IP of the connected [remote] machine to a name. Name resolution, particularly via DNS, may yield such a name.

Note: When specifying server lists, you must provide machine names [not FQDNs] in the configuration files and when acquiring an application from another machine.

Then configure name resolution to resolve names within the context of the appropriate domains by specifying domain suffixes during DNS configuration on the client machine [see diagram] and leave the mapping to DNS.
In the following diagram, a machine called XX will be searched for, firstly, within the trihedral.com domain and then in the xyz.com domain.



This means that machine names must be unique within the domains that RPC Manager can address, e.g. using the above configuration, RPC Manager will treat xx.trihedral.com and xx.xyz.com as the same machine [even if they are not].

Protocol

RPC Manager uses TCP/IP sockets to provide a transparent communication channel between RPC Manager instances. A proprietary protocol is transmitted between RPC Manager instances carrying both RPCs and control information.

This section describes the proprietary protocol used by RPC Manager. This information is not essential to use RPC Manager to its fullest capabilities.

Related Information:

...Protocol Versions - overview.

...General Structure

...Version 3 Packet Format

...Version 4 Packet Format

...Session Table Message

...Version 3 RPC Messages

...Version 4 RPC Messages

...Packed Parameters

Protocol Versions

There are two versions of the protocol, loosely named version 3 and version 4 (the version numbers actually refer to points of major change in RPCManager).
Version 3 protocol is used in all RPCManagers from VTS version 7.1.21 through version 9.
Version 4 protocol is used from version 10 onwards.
Version 4 has a different structure enabling RPCs from multiple source GUIDs and to multiple destination GUIDs to be transported in the same RPC packet, reducing the number of RPC packets needed (and hence the turnaround latency). Version 4 protocol supports packet compression and does so by default. Finally, VTScada TCP/IP engine changes permit version 4 to get increased throughput (particularly on WANs) by properly utilizing TCP window scaling and a larger default window size.
RPCManager versions using version 4 protocol also support version 3 protocol to enable support of mixed versions of VTScada in the same SCADA system.

General Structure

The RPC Manager protocol is a generic term that can be better defined as the cross-machine protocol used by two SocketNodes to provide the reliable RPC transport. By this definition, the protocol is private to SocketNode and is completely encapsulated within SocketNode.
TCP/IP provides for reliable delivery of packets of information, where, for the purposes of this document, a packet is defined as an atomically written block of bytes.
Each packet has the same format, information within the packet header being used to decode the message that it carries. Messages fall into two categories, control messages and RPC messages.

Version 3 Packet Format

A packet has the following format:

Sync	Dest GUID	Function Code	Message Length	Sequence Number	Source GUID	Message
4 bytes	16 bytes	1 byte	2 bytes	1 byte	16 bytes	variable

Sync

The Sync field is constant and is the character sequence #96#. As TCP guarantees reliable delivery, there is no CRC or checksum on a packet; this would be a duplication of the error checking abilities of TCP. The purpose of the Sync field is to provide defence against an internal RPC Manager fault, where a partial packet was transmitted. TCP guarantees reliable, atomic delivery of an atomically written sequence of bytes. The Sync field ensures that a non-atomic write (which should not happen) will not cause erroneous action at the receiving RPC Manager.

Dest GUID

A running VTScada system contains one "root" system object and multiple application "root" objects. The Dest GUID field is used to discriminate between these root objects and allow RPC Manager to

determine a starting scope in which to search for a target RPC sub-routine.
The Dest GUID for an application root is the same GUID as used in the SETTINGS.STARTUP file to uniquely identify an application's code objects within the distributed domain. That Dest GUID is the same for all instances of the same application.
RPCs directed at the system root level carry an all-zero Dest GUID. An all-zero Dest GUID is also used on control messages.

Function Code

The Function Code contains a unique value, defining the message type. This is combined with a set of modifier flags that provide for minor expansions to the "pure" set of commands.
The least significant 5 bits of the Function Code contains the command code. This gives a set of 32 possible commands. Command code 0 is, defensively, not used, giving a total of 31 commands, of which only five are defined:

Code Name	Value	Meaning
#FNF_PING	1	Ping packet. No message. The Binary GUID is all-zeros, the Message Length is zero and the Sequence Number is zero.
#FNF_ACK	2	acknowledgment of an #FNF_RPC. No message. The Binary GUID holds the same Binary GUID as the #FNF_RPC that is being acknowledged. The Message Length is zero. The Sequence Number contains the same Sequence Number as the #FNF_RPC that is being acknowledged.
#FNF_SESSION	3	Session management table in message.
#FNF_RPC	4	RPC in message.
#FNF_ RPCMULTI	5	Multiple RPCs in message.

The most significant 3 bits contain modifier flags. These flags are command specific and, presently, are only defined for #FNF_RPC:

Flag Name	Value	Meaning
# FNF_FLAG_ DUALGUID	32	The Source GUID is present.
#FNF_FLAG_FIRST	64	First fragment of a fragmented RPC.
#FNF_FLAG_NEXT	128	Subsequent fragment of a fragmented RPC

Message Length

The Message Length field contains the number of bytes that make up the Message Field.

Sequence Number

This is a number, initialized to 1, which increments each time a transmitted #FNF_RPC has been acknowledged. When it reaches 255 and increments, it wraps around to 1. For all other Function Codes, the number is zero.

The receiving SocketNode acknowledges, but does not action, an RPC message that was a duplicate of the previous one.

This is designed to prevent SocketNode taking replicate action if the same RPC is re-transmitted, as part of a retry strategy. In the situation where the remote machine sees the RPC message transmission and acknowledges it, but the transmitting machine fails to receive the acknowledgment, a successful re-transmission of the same RPC message will be acknowledged, but a second invocation of the RPC subroutine will not occur. This could happen because SocketNode will tolerate a high degree of inter-machine communication link disturbance, before session termination occurs.

If the session is lost, the Sequence Number is reset to 1.

Source GUID

The Source GUID is only present in cross-application RPC messages. It is the binary form of the GUID of the application that sourced the RPC.

Version 4 Packet Format

A packet has the following structure:

Function Code	Sequence #	Message Length	Uncompressed Length	Message
1 byte	1 byte	4 bytes	4 bytes	variable

Function Code

Version 3 protocol always started with a "Sync" marker of 4 bytes, the first byte of which was 0x23 (a # character). The version 4 function codes defines this function code value as a session table and interprets the remaining bytes of the transmission as an "old" format session table. When RPCManager opens a socket connection to a peer, this is the first message sent and enables the two RPCManagers to determine which protocol should be used. This avoids the need to have an old and new session table format.

The least significant 5 bits of the Function Code contains the command code. This gives a set of 32 possible commands. Command code 0 is, defensively, not used, giving a total of 31 commands, of which only five are defined:

Code Name	Value	Meaning
#FNF_PING	1	Ping packet (I have sent nothing for some time, but I'm still here). The entire transmission consists of only the function code.
#FNF_ACK	2	acknowledgment packet (acknowledging an RPC or RPCs received). The entire transmission consists of the function code and sequence number.
#FNF_SESSION	3	Unused. A version 4 format packet carrying this function code is treated as illegal and cause the session to reset.

#FNF_RPC	4	RPC in message. The entire transmission is of variable length and consists of the function code, sequence number, message length and variable length message field.
#FNF_ RPCMULTI	5	Unused. A version 4 format packet carrying this function code is treated as illegal and cause the session to reset.
#FNF_RPCCOMPRESS	6	Compressed RPC in message. The entire transmission is of variable length and consists of all the above fields.
#FNF_NEWSESSION	35	Session management table in message. Identical packet format to version 3.

Sequence Number

This is a number, initialized to 1, which increments each time a transmitted #FNF_RPC or #FNF_RPCCOMPRESS has been acknowledged. When it reaches 255 and increments, it wraps around to 1.

The receiving SocketNode acknowledges, but does not action, an RPC message that was a duplicate of the previous one.

This is designed to prevent SocketNode taking replicate action if the same RPC is re-transmitted, as part of a retry strategy. In the situation where the remote machine sees the RPC message transmission and acknowledges it, but the transmitting machine fails to receive the acknowledgment, a successful re-transmission of the same RPC message will be acknowledged, but a second invocation of the RPC subroutine will not occur. This could happen because SocketNode will tolerate a high degree of inter-machine communication link disturbance, before session termination occurs.

If the session is lost, the Sequence Number is reset to 1.

Message Length

This holds the number of bytes in the message field.

Uncompressed Length

This field, only present with function code #FNF_RPCCOMPRESS, holds the number of bytes in the uncompressed length of the message field.

Flags

The Flags byte contains routing flags that tell RPCManager how to process the message and flags that determine the format of the body:

Bit Number	Meaning
0 - 4	Routing Flags. Identical to those defined for previous versions of the RPC protocol, with the addition of the #FNR_EXSERVERS (2^4) flag for routing to potential server clients only.
5	Dual GUID. When set, a Source GUID field is present in the body. The Source GUID field is only present in cross-application RPCs.
6 - 7	Fragment Flags. This indicates:

Value	Meaning
0	The body is a fragment of an RPC spanning multiple bodies.
1	The body is the first fragment of an RPC spanning multiple bodies.
2	The body is the final fragment of an RPC spanning multiple bodies.
3	The body holds a complete RPC.

Session Table Message

A #FNF_SESSION packet is the first packet which is transmitted over a new socket stream connection. There is no acknowledgment that this packet has been received, however, there will be no transmission of any other packets until a #FNF_SESSION packet is received.

The message in a #FNF_SESSION packet contains the version number of the remote VTScada and a session management table:

Remote VTScada Version	Session Table Size	Session Table	Workstation Name	Flag Byte	Session Table Sequence #	VTScada Serial Number
8-byte IEEE floating point	4 bytes	variable	Text, NUL terminated.	1 byte	4 bytes	4 bytes

The Session Table Size is given in bytes and includes all bytes in the remaining fields.
The Session Table field holds the actual session table and is constructed from rows, each row forming the SID for a connection for an application:

Application GUID	Application Session ID	Connection Session ID
36 byte text GUID	36 byte text GUID	36 byte text GUID

This may appear to be a bandwidth-hungry message, but it is sent infrequently:

- As the first packet down a new socket stream connection.
- If an application terminates.
- If an application starts.

An operational system will probably only have two rows in the session table, one with an Application GUID field of the all-zero Binary GUID and one for the end-user application. Sending a row for the VTScada system GUID permits the detection of VTScada being restarted during a network break.
The Workstation Name is the NetBIOS name of the workstation sending the session table.
The Flag Byte holds bits that describe the capabilities of the sending RPCManager and what additional fields are present:

Bit Number	Meaning
0	The Session Table Sequence # field is present.
1	The VTScada Serial Number is present (VTS 9.0.06

onwards).

2 The sending RPCManager supports version 4 protocol.

The Session Table Sequence number is a 32-bit number, incremented each time a MachineNode sends a session table. This is used by the receiving RPCManager to detect session tables that arrive out-of-sequence on a multi-homed system. This prevents old session information from being interpreted as the current information where one route is slower than the other.

The VTScada Serial number is a 4-byte integer encoding of the VTScada serial number, exactly as obtained from a GetConfiguration(0) call.

Version 3 RPC Messages

#FNF_RPC

A #FNF_RPC message has the following, generic, format:

Mode Cut-Off	Routing Flags	Encoded RPC
1 byte	1 byte	variable

The Mode Cut-Off field contains the mode cut-off value that was supplied to the original \RPCManager\Send() call. The Routing Flags field contains the RPC routing flags, see section RPC External Routing.

If the Encoded RPC is longer than the value defined in the configuration variable RPCMaxPacketSize, the RPC will be broken into fragments for transmission, each fragment being no longer than RPCMaxPacketSize.

A fragmented RPC utilizes the two Function Code flags #FNF_FLAG_FIRST (2^6) and #FNF_FLAG_NEXT (2^7). #FNF_FLAG_FIRST is set on the first fragment and #FNF_FLAG_NEXT on all subsequent fragments. The #FNF_RPC messages carry an additional 8 byte header:

Block Number	Total Blocks
4 bytes	4 bytes

The Block Number field is set to zero and increments on each fragment, until Total Blocks minus 1 is reached. This denotes the last block in

sequence. The header is followed by the generic #FNF_RPC message, so an RPC that was fragmented over 3 packets would appear:

Syn-c/GUID	Func-tion Code (#FNF_RPC + #FNF_FLAG_FIRST)	Lengt-h	Sequenc-e (n)	Block Num-ber (0)	Total Block-s (3)	Mod-e Cut-Off	Rout-ing Flags	Encode-d RPC
Sync/GUID	Function Code (#FNF_RPC + #FNF_FLAG_NEXT)	Length	Sequence (n)	Block Number (1)	Total Blocks (3)			Encoded RPC
Sync/GUID	Function Code (#FNF_RPC + #FNF_FLAG_NEXT)	Length	Sequence (n)	Block Number (2)	Total Blocks (3)			Encoded RPC

#FNF_RPCMULTI

A #FNF_RPC message has the following, generic, format:

Length of this RPC	Mode Cut-Off	Routing Flags	Encoded RPC
4 bytes	1 byte	1 byte	variable

This message repeats throughout the length of the packet. Each message consists of an RPC for the same application GUID.

This function code was introduced in VTS version 5.18 as a means of more efficiently transporting updates for large numbers of drivers more efficiently. The encoding of the RPC is the same as #FNF_RPC.

RPC Encoding

The Encoded RPC field of the #FNF_RPC message has only one fixed length field, the RPC Code. All other fields are of variable length.

Target Type	Target	Module Name	Context	Packed Parameters

1 byte

The purpose of the Target Type and Target fields are to differentiate between directed RPCs and service RPCs. The Target field's type and content depends on the Target Type:

Target Type Field	Target Field	Description
Service RPC (0)	2 off 1 byte numerics	The first byte contains an offset into the Context field where the service name can be found. The second byte contains the length of the service name.
Service RPC (1)	CR terminated string	The target field contains the service name, terminated with a carriage return.
Directed RPC (2)	CR terminated string	The target field contains the machine name or IP, terminated with a carriage return.

The Module Name and Context fields are also strings terminated with a carriage return.

The Packed Parameters field is described in the section "Packed Parameters" and is a common encoding with version 4 protocol.

Version 4 RPC Messages

#FNF_RPC

The message field consists of one or more "elements". An element contains either a complete RPC or a fragment of an RPC. The message may contain any combination of these. Elements are interpreted in the strict order that they appear in the message, from first received to last received.

The format of an element is

RPC Message Header	Packed parameters

variable	variable

The header description follows.

The Packed Parameters field is described in the section "Packed Parameters" and is a common encoding with version 3 protocol.

RPC Message Header

Flags	RPC Length	Mode Cutoff	Dest GUID	Source GUID	RPC Type	Service Name or IP/Machine	Module Name	Context	Fragment ID
1 byte	4 bytes	1 byte	16 bytes	16 bytes	1 byte	CR terminated	CR terminated	CR terminated	4 bytes

Flags

The Flags byte contains routing flags that tell RPCManager how to process the message and flags that determine the format of the body:

Bit Number	Meaning
0 - 4	Routing Flags. Identical to those defined for previous versions of the RPC protocol, with the addition of the #FNR_EXSERVERS (24) flag for routing to potential server clients only.
5	Dual GUID. When set, a Source GUID field is present in the body. The Source GUID field is only present in cross-application RPCs.

6 - 7 Fragment Flags. This indicates:

Value	Meaning
0	The body is a fragment of an RPC spanning multiple bodies.
1	The body is the first fragment of an RPC spanning multiple bodies.
2	The body is the final fragment of an RPC spanning multiple bodies.
3	The body holds a complete RPC.

Mode Cut-off

The Mode Cut-Off field contains the mode cut-off value that was supplied to the original \RPCManager\Send() call.

Dest GUID

This field is the same as the version 3 protocol header field of the same name and contains the application GUID that the RPC is intended for, with an all-zero GUID conventionally meaning the system root.

Source GUID

This field is optional and only present if the Dual-GUID flag is set in the Flags byte. If present it contains the application GUID that sourced the message. If absent, the sourcing application GUID is the same as the Dest GUID.

RPC Type

This field indicates if the RPC is a service (1) or a directed (2) type

Service Name or IP/Machine

This field contains the service name, if the RPC is a service RPC or the target machine name if the RPC is a directed RPC.

Module Name

This field contains the name of the module to be invoked as the target of the RPC.

Context

This field contains the scope in which to find the module to be invoked. This is specified as a scope string, with the starting point of the scope being determined by the Dest GUID and Service Name (if a service RPC).

Fragment ID

This field is optional and only present if the RPC is fragmented. If the RPC is fragmented the Flags byte will not have both the first and last fragment flags set. A complete RPC will have both flags set and will not have a Fragment ID field.

The fragments are concatenated together at the receiver and processed once the final fragment is received. The existing algorithm in the SocketNode receiver is used to maintain the stream of fragments in a temporary disk file stream, rather than a memory stream, once the fragment concatenation becomes too large.

Packed Parameters

The parameters to the RPC subroutine are packed into the Packed Parameters field. The Packed Parameters field is simply the output of Pack. Running this field, in its entirety through UnPack() results in exactly the same data structure that was passed into Pack() when the RPC was encoded.

Type	Range	Encoded As
Invalid or any unsupported type		Single byte of value 255.
Numeric Integer	0 to 0xDF	Single byte value.
Numeric	0xE0 to 0xFFF	A byte of 0xE0 or'ed with the top 4 bits of

Integer		the numeric, followed by the least significant byte of the numeric.
Numeric Integer	0x1000 to 0x7FFF	A byte of 0xF0, followed by the 2-byte numeric.
Numeric Integer	0x8000 to 0xFFFF	A byte of 0xF1, followed by the 2-byte numeric.
Numeric Integer	0x10000 to 0x7FFFFFFF	A byte of 0xF2, followed by the 4-byte numeric.
Numeric Floating Point	IEEE 4-byte floating point number	A byte of 0xF3, followed by the 4-byte IEEE floating point number.
Numeric Floating Point	IEEE 8-byte floating point number	A byte of 0xF4, followed by the 8-byte IEEE floating point number.
Text		A byte of 0xF5, followed by a 2-byte length of the text, followed by the text.
Stream		A byte of 0xF6, followed by a 4-byte length of the stream, followed by the stream.
Array		See following notes about Packed Arrays.

Packed Arrays

The first byte of an array encoding consists of a byte of 0xF7 or'ed with the number of array dimensions minus 1. There then follows a table, with one row per array dimension:

Start Index	Number of Elements

Each table row is packed in exactly the same manner as given for scalar values in the preceding section, minimizing the size of the table.

Each dimension of the array, in ascending order then has its elements packed, in ascending order, in the same manner as scalar values. This operates up to a maximum of three dimensions.

Related Functions:

... Pack

... Unpack

Security Manager

The Security Manager is a basic component of VTScada supplying all of the code necessary to provide varying levels of security in an application. The Security Manager is available for use with both Script and Standard applications. It provides a set of API functions to create and modify user accounts, verify access credentials and query relevant permissions. It also provides a set of public variables that may be examined by application code.
User accounts are normally stored on an application-by-application basis. You can enable Shared Security in an application in order to use the security database of its OEM layer.
The Security Manager is always active.

Related Information:

...Accounts - User Accounts

...Roles - Security Roles

...Security Rules - Privileges granted to Roles or Accounts

...Security Implementation - An overview of the Security Manager privilege system

...The SecurityManager API - Structures, functions and publicly accessible properties.

...Security Event Logging

...Security NameSpaces - Account configuration for Realm Area Filtering

---

Accounts

There are two types of account: the user and the role. User accounts identify each person using the application, including their name, password, privilege set and other security-related information. A role is a named collection of security rules. One or more roles may be assigned to

each user account, thereby simplifying the process of assigning privilege rules.
Both user accounts and role accounts are stored using the same data structure, within the Accounts.Dynamic file, which can be found in the application directory.
In most cases, accounts will be created by way of the user interface in the Accounts dialog. You may also add, modify and delete accounts through code, noting that there are security restraints in place on these functions to prevent unauthorized tampering.
While accounts are commonly referred to by name, each account is actually identified by an ID code. You may therefore change the name of an account if needed without losing any of its configuration.
See also: AccountData Structure, Account Storage, Security Rules.

Related Information:

...Account Storage

...Alternate Identification

...Roles

Account Storage

The Security Manager database is held in a file within the application directory named Accounts.Dynamic. The information in this file cannot be read or successfully modified by any means other than the Security Manager user interface.
In the case that Shared Security is in use, the Accounts.Dynamic file of the OEM layer will be used for all applications based upon that layer.
There are three sections in the Accounts.Dynamic file:
[SecMgr] Has only one entry: the version number of the security manager database. This will be "140" for all versions.
[Accounts] Stores the information for each user account, in the form used by VTS version 10.
[Accounts-200] Stores the information for each user account, in the form used by VTScada after version 10.0. For applications upgraded from

version 10, VTScada will read the information in the [Accounts] section once, then create an [Accounts-200] section containing the same information stored in the new format. After the [Accounts-200] section has been created, VTScada will no longer read the [Accounts] section
Each entry in the database takes the form, "AccountID = Encoded account definition". Restrictions are in place so that the Accounts.Dynamic file from one application cannot be used in another, nor can an account definition from one application be copied for use in another application.

Alternate Identification

Alternate identification for an account may be assigned. The identification is typically a numeric code that operators may use when logging in to the Alarm Notification System.
No two users may have the same alternate ID - each must be unique. The minimum length of the ID is controlled by the Setup.INI property, MinAltIDLength, although a default minimum of four characters will be used if this property has not been set. MinAltIDLength must be added to the [SYSTEM] section.

Note: Trihedral Engineering strongly discourages any reduction in the number of characters required for identification. Any reduction in the number of possible combinations makes it easier for an attacker to compromise your system.

Another possible alternate identification source is a card reader. If your application uses one of these devices, a custom module will need to be written to handle the communications. Please contact Trihedral for support.

Related Information:
See the VTScada Developer's Guide for:

...Configure Alternate Identification

...Proximity Card Readers

Roles

A role is a type of account. The purpose of a role is to associate a set of security privileges or rules to a name that can then be associated with user accounts. Roles are meant to define job functions such as "Operator" and "Manager", with all of the security rules required for that job. User accounts may be assigned multiple roles, gaining the combined privilege sets of all.

This association between a role and a user remains dynamic so that if changes are made to a role's privilege set, all users who have been assigned that role will immediately have their privileges changed.

While a Role is a type of account, roles differ from user accounts as follows:

- Roles do not have passwords.
- Roles do have descriptions.
- It is not possible to log in to an application using a role name.
- Automatic time-out periods do not affect roles.
- Roles do not have alternate ID values.
- Roles are managed in an area of the Accounts dialog that is separate from the list of user accounts.

A user account can be used as the template for a new role, which will then have all the same privileges, but no continuing link. The reverse is also true.

Related Information:

...The Logged Off Role

The Logged Off Role

A role named "Logged Off" will be found in every installation as soon as security is activated. This role is in effect when no other user is logged in. You may use this role to permit a privilege for unrestricted access. For example, you might allow anyone to view the alarm page without first logging in. Take care not to open a security hole in your application by granting unnecessary privileges to this role.

You cannot delete the Logged Off role.

Note: The Logged Off account has a second use in VTScada which is to determine whether VIC sessions would remain active when the logged off. If the role has a password, then VIC sessions will remain active. This function is now handled by a check box in the Administrative Settings user interface, but you should use care if modifying the Logged Off account through code.

Security Rules

Prior to VTS version 10.1, resources in an application were secured using privileges. A privilege was created, then assigned to an output tag or page. The privilege could then be assigned to a user so that they would have permission to write data with the output tag or open the page. This led to a coarse-grained security model where the only way to increase the granularity of security was to assign new privileges.
VTS version 10.1 introduced the concept of a security rule. Rules are assigned to accounts, not to resources. A rule is composed of three parts:

- A mandatory privilege number (or a role name, thereby including a set of privilege numbers).
- An optional scope.
- An optional workstation name.

A scope is the name of a tag. A rule containing a scope applies to that named tag and all children of that tag. For example, if a tag is named Tag1\Tag2\Tag3 and a scope of Tag1\Tag2 is specified in a rule, that rule will apply to Tag1\Tag2 and all its children including Tag1\Tag2\Tag3.
When a check is made to determine if the current user has access to a resource, the privilege number assigned to the resource, the name of the tag associated with that resource and the workstation on which the request is being made are compared against the set of security rules

assigned to the user. For a check to pass, all component parts of one of the user's security rules must match.
Therefore, a rule that has only a privilege number is a "global" privilege – it will match any resource that requires the same privilege number.
A rule that has a privilege number and a scope will only match if the privilege numbers match and the name of the resource or any parent of that resource match the rule's scope.
A rule that has a privilege number, a scope and a workstation name will only match if the privilege numbers match and the name of the resource or any parent of that resource match the rule's scope and the request is being made on the same workstation as specified in the rule.
This permits the specification of finer-grained security without the need to expand the set of privileges significantly. The concept can be thought of as a privilege number defining a verb describing the operation (e.g. "control" or "view") and the scope defining a noun on which the verb operates (e.g. "TreatmentPlant"). The Workstation modifier adds a location clause, referring to the operator (e.g. "from OfficeComputer"). If the tag database is organized into a hierarchy, the noun can encompass a collection of tags related by the hierarchy.
Conversion of a 10.0 or earlier VTS security database results in an equivalent set of global rules being generated for each user account.
SecurityCheck determines the tag name of the resource being checked. It searches the caller and its parent scope(s) for the name of a tag by looking for a "Name" variable. When it finds one, it looks up the VTSDB to determine if the value of the found Name variable is indeed the name of a tag. If not, the search continues. If so, the discovered name is used to check the user's security rules. This search can be bypassed by passing in a valid string to SecurityCheck's TagName parameter.
If no name is supplied nor can be discovered, only global rules will be able to match the SecurityCheck request, as all parts of a rule must match for a check to be successful.
Widgets and any code called by them that calls SecurityCheck therefore must be careful to ensure that they either supply a TagName parameter

to SecurityCheck or that they call SecurityCheck from a child scope of a tag. As most tag-centric widgets do run in tag scope, this is unlikely to be an issue in your code.

Related Information:

...Combining Security Roles and Rules

...Accounts

...Roles

Combining Security Roles and Rules

As described in the topic Roles, a role is a security account whose purpose is to encompass a named set of security rules. The intent is to provide an easy way to represent a set of commonly assigned security rules. For example, a role named "operator" may have a set of security rules that are commonly assigned to any user that performs the tasks of a plant operator.

A security rule can reference a role instead of a privilege number, thereby allowing a role to be assigned to a user account. As a role is an account, a role can also contain a security rule that references another role, thereby allowing a role such as "engineer" to contain the "operator" role.

Because a security rule has an optional scope and workstation name, a security rule that contains a role can also qualify the role using a scope or workstation name. This enables you to define rules such as an operator for a particular plant area ("operator" - "TreatmentPlant") and even enables you to restrict the workstation at which that rule can be effective.

Where a security rule specifies a role and either or both the optional scope and workstation name, the scope is only applied to security rules within the role that do not explicitly define a scope and, likewise, the workstation name is only applied to security rules within the role that do not explicitly define a workstation name.

Where a role contains a security rule that references another role, scope and workstation name overrides are applied recursively, as described above.

Security Implementation

For every application, the security manager is activated when the application is activated. Applications are activated when an operator requests some action of them (for example, attempting to access the Application Configuration dialog) or, in the case of OEM layers, when a dependant application is activated.

The DisplayManager, the LayerModule and each VIC session all have the concept of a user security session. SecurityManager maintains state information for each user security session via the call tree of any code that makes an API call into SecurityManager.

Making a call from code within that call tree causes SecurityManager to use the security rules for the call tree's user security session when evaluating permissions.

Making a call from outside such a call tree causes SecurityManager to use the Logged Off role security rules.

If the application is in a secured state, the Security Manager provides a number of variables controlling how security is managed and a set of functions for working with user accounts and checking access privileges.

Related Information:

...System Privilege Reference for Programmers

...Application Privileges

...Shared Security

System Privilege Reference for Programmers

The following is a list of system privileges for the current implementation.

Constants must be preceded by \SecurityManager\ unless you have imported the API as described in The SecurityManager API.
System privilege numbers are <= 0. Application privilege numbers are >= 16. The table is complete; missing values are deprecated privileges.

System Privilege	Constant	Value	Description
Administration			
Security Administrator	PrivBitAdministrator	-4	Permits access to the Administrative Settings dialog and modification of administrative functions. Also required to modify security roles.
Configure	PrivBitConfigure	0	Permits access to the Application and Configuration dialog, the Import File Changes button on the VAM and the right to delete applications.
Account Control			
Accounts Manager	PrivBitManager	-3	Permits manipulation of the Account List; allows the user to add, copy, delete, and modify user accounts.
Account Modify	PrivBitAccountModify	-2	Allows users to modify their own password, but does not allow

			them to modify their account privileges.
Account View	PrivBitAccountView	-1	Allows users to view (but not modify) their own privileges.
Internet Client Access	PrivBitInternetClient	-23	Allows users to make connections to a VTS/IS using a VIC.
Application Control			
Application Stop	PrivBitAppClose	-9	Allows users to stop the application
Application Manager View	PrivBitVAMView	-40	Allows users to view the VAM when the Setup.INI property, HideWAM, is set to TRUE.
Version Control			
Advanced Version Control	PrivBitVersionControl	-39	Allows a user to switch or revert versions in the Version Log.
Deploy Changes	PrivBitDeploy	-15	Allows users to perform updates through the Application Configuration dialog.
Revert Changes	PrivBitRevert	-16	Allows users to perform rollbacks through the Application Configuration dialog.

Application Configuration			
Edit Files	PrivBitEditFiles	-14	Allows users to change files through the Application Configuration dialog. Also required for the Compile button on the VAM. (Formerly called "Remove File")
Page Add	PrivBitPageAdd	-17	Allows users to add pages through the Idea Studio.
Page Modify	PrivBitPageModify	-18	Allows users to modify page properties through the Idea Studio.
Page Delete	PrivBitPageDelete	-19	Allows users to delete pages through the Idea Studio.
Page Note Edit	PrivBitPageNoteEdit	-37	Allows a user to add, edit or delete page notes.
Page Note Hide	PrivBitPageNoteHide	-38	Allows a user to make page notes hide without deleting them.
Tag Operations			
Parameter View	PrivBitParamView	-11	Allows users who do not have the Tag Modify privilege to view tag parameters.

Tag Add/Copy	PrivBitTagAddCopy	-20	Allows users to add or copy tags through the Tag Browser. Tag Modify also required.
Tag Modify	PrivBitTagModify	-21	Allows users to modify tag properties through the Tag Browser.
Tag Delete	PrivBitTagDelete	-22	Allows users to delete tags through the Tag Browser.
Manage Tag Types	PrivBitManageTagTypes	-41	Allows use of "Create new type" and "Redefine type" in the Tag Browser. Allows use of "Manage Types" in the Application Configuration dialog.
Manual Data	PrivBitManualData	-6	Set or change the Manual Data value of a tag without having the Tag Modify privilege.
Questionable	PrivBitQuestionable	-7	Change the Questionable flag of a tag without having the Tag Modify privilege.
Alarm Operations			
Alarm Acknowledge	PrivBitAlarmAck	-8	Allows operators to acknowledge alarms.
Alarm Disable	PrivBitAlarmInhibit	-5	Allows operators to disable alarms.

Alarm Mute	PrivBitAlarmMute	-24	Allows users to use the Mute button on the Alarm page to mute all current and future alarms.
Alarm Silence	PrivBitAlarmSilence	-25	Allows users to use the Silence button on the Alarm page to silence the sounding alarm.
Alarm Shelve	PrivBitAlarmShelve	-42	Enables operators to shelve alarms, leaving them enabled but deactivating all notifications.
Historical Data			
Group Delete	PrivBitHDVGroupDelete	-28	Allows the user to delete pen groups for the Historical Data Viewer page.
Group Modify	PrivBitHDVGroupModify	-26	Allows the user to modify pen groups for the Historical Data Viewer page. If denied, then Group Delete and Group Save are also effectively denied.
Group Save	PrivBitHDVGroupSave	-27	Allows the user to save pen groups for the Historical Data Viewer page.
Note Add	PrivBitNoteAdd	-30	Allows the user to add

			notes to a notebook tag using the Historical Data Viewer page.
Pen Modify	PrivBitHDVPenModify	-29	Allows the user to modify pen properties for the Historical Data Viewer page.
Page Access			
Alarm Page Access	PrivBitAlarmPageAccess	-31	Allows the user to access the Alarm page.
History Page Access	PrivBitHDVAccess	-33	Allows the user to access the Historical Data Viewer page.
Internet Client Tools Access	PrivBitVICTools	-34	Allows the user to access the debugging and analysis tools included with VTScada (see "Debugging and Analysis ").
Internet Client Monitor Access	PrivBitVICMonitorView	-35	Allows a user at a VTScada internet client to view the internet client monitor page.
Internet Client Monitor Admin	PrivBitVICMonitorAdmin	-36	Allows a user at a VTScada internet client to operate the internet client monitor page.
Reports Page Access	PrivBitReportsPageAccess	-32	Allows the user to access the Reports page.

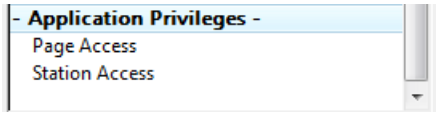
Three other constants are defined, which are duplicates of values in the above table. These exist for backward compatibility.
PrivBitRemoveFile == PrivBitEditFiles
PrivBitUpdate == PrivBitDeploy
PrivBitRollback == PrivBitRevert

Application Privileges

Application Privilege are those that developers create for a given application. They are generally used to restrict access to custom pages and output tags. While system privileges control such actions as acknowledging alarms and adding pages, it is the job of Application Privileges to restrict access to developer-created pages and writing to output tags.
Every application privilege will have an index number, starting at 16. Also, every application privilege is enumerated in the configuration file, Settings.Dynamic with values starting at zero. When writing expressions that check privileges, add 16 to the enumerated value in Settings.Dynamic.
The first variable in this section, PrivBitsTotal, is a count of the current number of application privileges. For each privilege, the number following the name controls the order in which the privileges will be displayed in the user interface.
Separator lines for the user interface are stored using the format, "PrivSepDesc0 = -- Description, 1".
For example, the following set of privileges:

```
<SECURITYMANAGER-PRIVAPP>
  PrivBitsTotal                    = 2
  PrivDesc0                        = PageAccess,0
  PrivDesc1                        = StationAccess,1
```

Matches this set in the Administrative Settings dialog.



Shared Security

Some sites run more than one application, where those applications are based on a common OEM layer. In this situation, managers may want to create a single security database and share it between those applications, rather than maintain security accounts and settings for each application. The Shared Security feature allows this.

Shared Security is enabled in the application layer by using the Administrative Options dialog to select a security provider database other than the current application's. Only OEM layers will be available in the selection.

User accounts may be configured in any of the applications sharing a database, but will be stored only in the OEM layer's Accounts.Dymanic file and will apply to all applications based on that layer. Any pre-existing information in an application's Accounts.Dynamic file will be ignored after shared security is enabled.

Note that the fundamental OEM layer, VTScada, cannot be selected as the security provider.

If using Shared Security, it is important that only one application be running a security alarm module – see the note in Security Plug-in Modules for SecAlarm.

The SecurityManager API

The SecurityManager API definitions can be imported into your application by calling:

```
\System\ImportAPI(\SecurityManager);
```

The effectively includes the definitions in the calling module allowing you to omit the \SecurityManager prefix when accessing these variables. For example:

```
\SecurityManager\PrivBitManager
```

can be simply written:

```
\PrivBitManager
```

You should check the return value of ImportAPI. It returns the number of variable name clashes that occurred when the import was attempted. Zero is therefore a successful result.
Note that the following restrictions are in place:

- The API calls will use the security context of the caller to verify that the user has Manager privilege. Any calls made from a user that is not so privileged will fail and a security event will be logged.
- The API calls only operate on the security database associated with the security context of the caller, thereby preventing interference from another application or system level code.

Related Information:

...AccountData Structure

...SecurityRule Structure

...Security Manager Return Codes

...Security Manager Functions

...Security Manager Public Variables

...Security Plug-in Modules

AccountData Structure

Information about each account (both user accounts and roles) is stored in the following structure:

```
    AccountData Struct [
      AccountID                   { Unique ID of this account
};
      AccountName                 { Unique name of this account
};
      Password                    { Password - only used for user
accounts    };
      AltID                       { Alternate ID - user accounts only
};
      AutoLogoff                  { Automatic log-off timeout - user
a/c only };
      PWDate                      { Password creation date - user a/c
only    };
      Rules                       { Array of SecurityRule structures
};
      IsRole                      { TRUE if account is a role, else
```

```
user      };
      Disable                    { TRUE to disable this account
};
      Description                { Textual description of this
account       };
      CustomData                 { Uncommitted field for application
data    };
    ];
```

API module calls that require an AccountData structure for an existing account must provide a valid AccountID. An AccountID is a text value whose length is specified in the imported API constant AccountIDLength.

AccountName	Holds the unique name of the account, including any namespace (group) prefix, separated by your application's configured NameSpaceDelimiter character. You may change an account name, which is why the API requires the immutable AccountID for all operations on an existing account.
PasswordUser	accounts only. Any supplied passwords must conform to application configured password strength requirements.
AltID	User accounts only. This is the alternate account identification used by such subsystems as the Alarm Notification System.
AutoLogoff	User accounts only. It specifies the time, in minutes, after which a user session using this account will be logged off if there is no UI activity. This overrides the application configured global AutoLogoff value.
PWDate	User accounts only. This is the date on which the current password was created. It is used to enforce password change after a period of time. When creating an account, an Invalid value automatically sets the PWDate to today. If set to zero, it forces the user to change their password when they next log in.
Rules	An array of SecurityRule structures, one per rule. If Invalid, the user or role has no privileges whatsoever.
Disable	A Boolean value, defaulting to FALSE. If set to TRUE, the account is disabled. If this is a user account, the user cannot log in and, if already logged in, is immediately logged out. If this is a role, its security rules are disabled.

Description	Role accounts only. The purpose is to provide a meaningful description of the purpose of a role. Defaults to Invalid.
CustomData	Not used by SecurityManager. It is provided for application use to store any account-specific data it chooses. The data must either be text or numeric. If you need to store more complex data, serialize it into text before storing.

See: SecurityRule Structure.

SecurityRule Structure

The SecurityRule definition is used by the API to represent the content of a security rule. The structure is part of the imported API. It has the following format:

```
    SecurityRule Struct [
      PrivRole                { Privilege number or role account ID
};
      TagName                 { Name of a point in the tag tree
};
      Workstation             { Name of a workstation for this rule
};
      Disable                 { TRUE to disable the rule
};
    ];
```

PrivRole — Must be valid and contain any of

- A system-defined privilege number (examine the PrivBit… constants in the imported API).
- An application defined privilege (defined in your Settings.Dynamic file at starting at a value of 16, for historical reasons)
- The AccountID of a role or the name of a role. If a role name is specified, SecurityManager will convert this to an AccountID. Even if the role does not yet exist, it will convert this to an AccountID when the role becomes valid.

TagName — May be Invalid, in which case it does not play a part in the rule evaluation. Otherwise, this is the name of a tag, defining the scope in which the rule will permit access.

Workstation	May be Invalid, in which case it does not play a part in the rule evaluation. Otherwise, this is the name of a workstation that the account must be logged in at in order for the rule to permit access.
Disable	Defaults to FALSE. If set to TRUE, the rule is disabled. Any logged-on accounts using this rule will immediately have this rule disabled.

Security Manager Return Codes

The account manipulation methods all provide a return code that is one of the #SMAPIErr values defined in the imported API:

```
Constant #SMAPIErrSuccess     = 0 { Successful result
};
Constant #SMAPIErrNoUser      = 1 { User doesn't exist
};
Constant #SMAPIErrNeedManager = 2 { Caller did not have Manager priv-
ilege };
Constant #SMAPIErrUserExists  = 3 { User already exists - can't add
again };
Constant #SMAPIErrBadParm     = 4 { Bad parameter
};
Constant #SMAPIErrNotEditable = 5 { The application is not editable
};
Constant #SMAPIErrPwdTooWeak  = 6 { Password is too weak
};
```

Note that GetAccountInfo (described in the Query Module Calls) is useful in obtaining account information in the same format as is required by ModifyAccount and DeleteAccount. The error code can be converted to text e.g. for error display purposes by calling UIErrorToText().

Security Manager Functions

Four modules make up the Security Manager API, each containing its own set of function calls. These are:

Account Manipulation Methods

AddAccount	Creates a new account.
ModifyAccount	Modifies an existing account.
DeleteAccount	Removes an account.

Query Module

SecurityCheck	Examines the rules that apply to the current user or the named user to determine if the specified privilege has been granted.
BuildFullName	If a namespace and namespace delimiter are being used, returns the full, namespace-qualified name of the specified account.
GetFullName	Returns the full, namespace-qualified name of the caller's account.
GetGroupName	Returns the namespace of the caller's account.
GetUserName	Returns the user name of the caller's account.
GetAccountID	Returns the account ID of the named account.
GetAccountInfo	Returns one or more AccountData structures.
IsLoggedOn	Returns TRUE if the calling user is logged on, else FALSE.
IsSecured	Returns TRUE if the application has any user accounts defined, else FALSE.
IsSuspended	Returns TRUE if the user's account is suspended, else FALSE.
UIErrorToText	Returns a text string corresponding to the error code provided.

VTScada Authentication Module

AlternateIdCheck	Searches the accounts for an account whose AltID matches the parameter value.
AlternateLogon	Either creates, or attempts to log in using an alternate ID value. See comments.
AlternateLogoff	Synonym for LogOff().
Authenticate	Authenticates the Namespace, UserName and Password.
QuietLogon	Authenticates the authorization token (AuthToken) and, if successful logs the calling user session on as the user specified in the AuthToken.

LogOff	Logs the calling user session off.
UserCredChange	The return value will increment each time there is a change in the user session's logged-in user or their password.

Windows Authentication Module

UserLogonDialog	Returns the string value of the LDAP default naming context for the host machine domain.
WindowsLogon	Authentication request to Windows Authentication services.

User Interface Module

UserLogonDialog	Launches the Logon dialog.

Security Manager Public Variables

The following public, read-only variables are available for applications to examine:

InitComplete	Boolean. Set TRUE when you can call APIs that do not require accounts and settings to be loaded. Intended for system internal use only.
Ready	Boolean. Set TRUE when you can call APIs that do require accounts and settings to be loaded. Intended for general use.
Started	Same as Ready. For backwards compatibility.
SecMgrStatus	Artificial RPC status value for backwards compatibility. Always set to #RPCServer (2), once Ready is TRUE.

Security Plug-in Modules

Plug-in modules allow users to override default behaviors in VTScada applications. To override these modules, users can declare replacement modules in the PLUGINS section of the VTScada application's AppRoot.src source file.
The Security Manager recognizes two plug-in modules:

SecAlarm (SecAlarm.src)

The SecAlarm plug-in is responsible for alarm generation as prompted by security-related events.

> Note: If you are using Shared Security across several applications and you have provided an override for the SecAlarm plug-in module in one of the applications, you must create a dummy SecAlarm plug-in for use in the other applications. To avoid race conditions or other conflicts, only one of the applications should be running a security alarm module. Security events for all the applications sharing security will be logged only in that one application running the plug-in.

SecDenied (SecDeny.src)

The SecDenied plug-in is responsible for displaying the feedback dialog that appears when a security check fails.

Security Event Logging

Information is logged on every security-related event witnessed by the Security Manager, including (but not limited to):

- Logon/logoff events
- Account manipulation events

This information is used mainly as a form of feedback. You may view this logged information in the History list on the Alarm page.

Note that, in the case of Shared Security, logging takes place only in the application that runs the security alarm module.

Related Information:

...Shared Security

Security NameSpaces

VTS enables you to subdivide security accounts into name spaces. When namespaces are in use, the Logon dialog will query users for their group name as well as their user name and password. Namespaces are therefore sometimes referred to as security groups.
A given namespace can be associated with one or more tag Area properties. The result is that the users belonging to that security group will only be able to access the tags belonging to the assigned areas. This functionality can be organized using security name spaces in combination with realm area filtering (security name spaces on their own are not sufficient to segregate user data). For example, you may use security name spaces and realm area filtering together in applications where you must restrict sets of users to specific sets of pages or subsets of data, and where managers or administrators must be able to oversee their own user base, but should be unaware of any other end-users.
Two variables in the Settings.Dynamic <SECURITYMANAGER-ADMIN> section are associated with security name spaces. These are:

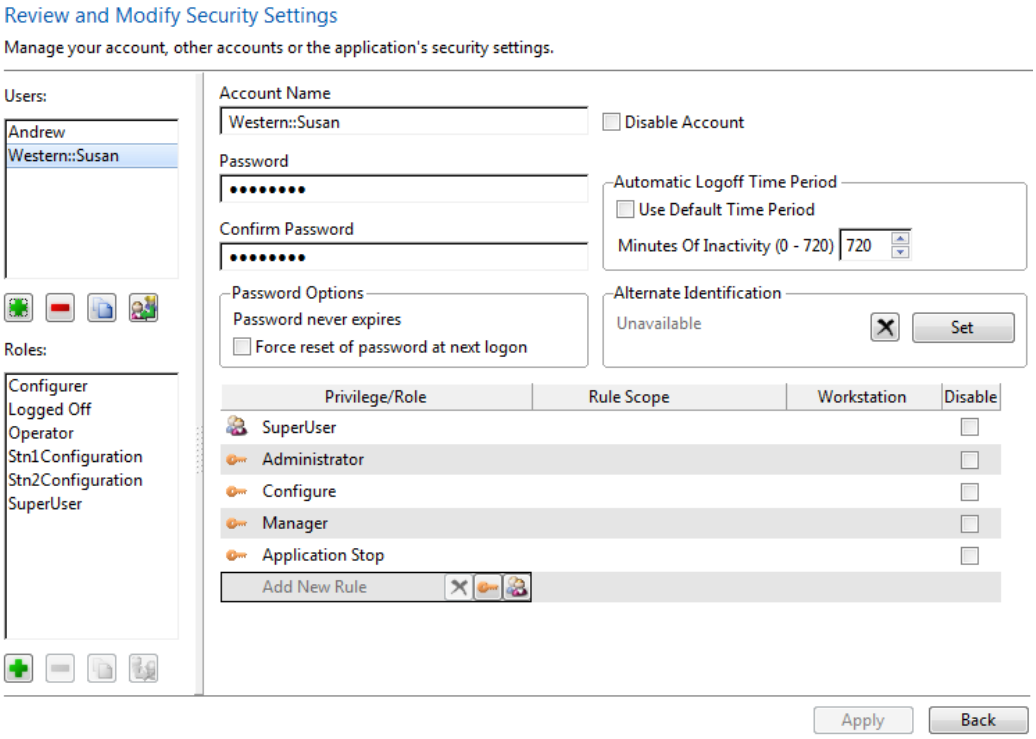
```
<SECURITYMANAGER-ADMIN>
NameSpaceDelimiter =
GroupLogin =
```

Set GroupLogin to 1 to enable group logins, and NameSpaceDelimiter to one or two characters that will be used as the delimiter. A colon ":" is commonly used as the delimiter.

NameSpaceDelimiter

The NameSpaceDelimiter application property enables you to specify the character (or characters) you wish to be used by managers setting up security groups. The recommended characters for NameSpaceDelimiter are two colons; however, you may use any characters you deem appropriate. The assigned character (or characters) must then separate the name of the security group from the username of the user belonging to that group when a new security account is added to your application.

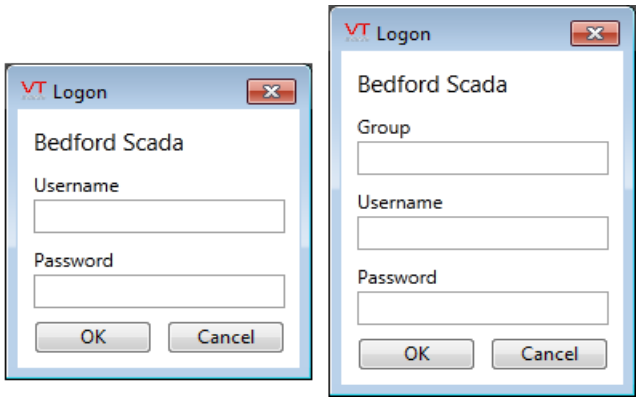
The following image displays the Add Account security dialog when a group is being specified for a new user. As you can see from this example, the double-colon has been assigned as the NameSpaceDelimiter.



GroupLogin

The GroupLogin application property enables you to add a third field to the Logon dialog that opens when the Logon button in the Display Manager's title bar is clicked. This third field is the Group field, which enables users to specify the group to which their user account belongs when they logon to an application. To include the Group field in the Logon dialog, you must set GroupLogin to 1.

The following image on the left displays the Logon dialog when GroupLogin has been set to 0 (its default value), while the following image on the right displays the Logon dialog when GroupLogin has been set to 1.



When logging on, users must enter the name of the security group to which they have been assigned in the Group field.

A super user (one who has not been assigned to any group) may leave the Group field blank, and can logon as they normally would, by entering their username in the Username field, and their password in the Password field.

Super users cannot log in via the VIC unless extra configuration is done as follows:

- The RootNamespace Settings.Dynamic variable has to be set to a value for the super user realm, which is different from all other configured realms.
- That realm must be configured in the Internet settings dialog with the required application listed.

The URL used to log in becomes "http[s]://servername/superrealm" where superrealm is the name assigned to RootNamespace and configured in Internet settings. A super user can then log in using their (non-namespace) username and password

Socket Server Manager

This service is designed to manage inbound TCP/IP and UPD/IP sockets that are shared by multiple drivers. It was modeled after the Modem Manager and uses the same discriminator driver interfaces. Inbound socket streams are passed to this service so that the stream will be given to the appropriate driver instance.

Related Information:

...Socket Server Manager - Error Logging

...Socket Server Manager API

Socket Server Manager - Error Logging

Errors are logged to a text file named "SocketServerEvents.log". Two application properties are available to control what is logged:
\SSMLogConnectFail When set to TRUE, the manager will log failed connection attempts.
\SSMLogConnectSuccess When set to TRUE, the manager will log successful connection attempts.
Both properties will default to FALSE.

Socket Server Manager API

The SocketServerManager was modeled after the Modem Manager and uses the same discriminator driver interfaces.
Inbound socket streams are passed to this service so that the stream will be given the appropriate driver instance.
There is one publicly accessible variable: SocketServerManager\Started will be true when the service has started.

Related Functions:

SocketServerManager\ArrayToString

Description: Utility function to pack an array of filtering addresses into a single string. The elements will be delimited using semi-colons.

Returns: Text

Usage: Script

Function Groups: Stream and Socket

Format: \SocketServerManager\ArrayToString(AddressArray)

Parameters:

AddressArray

Required. An array of IP addresses, where each array element is a text value.

Comments: None.

See Also:

... SocketServerManager\StringToArray

SocketServerManager\Register

Description Register a station with a group.

Returns Invalid when complete

Usage Script

Function Groups Stream and Socket

Format \SocketServerManager\Register(Context, StationKey, GroupName[, IPAllowString])

Parameters

Context

Required. The context should be the root tag. This must include Discriminator() and Context() sub-routines.

StationKey

Returned by Discriminator() in the root tag. A unique string that has meaning only VTScada, used to identify a particular station.

GroupName

The name of an IPListener tag (or group). This will be the source of the new data streams.

IPAllowString

An optional string. If valid, inbound connections will be restricted as specified. May be a partially specified address in order to allow a range of connection IP addresses. The string is one or more semicolon delimited IP addresses (or address ranges)

Comments

Public subroutine used by modules that are to receive inbound connections. This is intended for, but not limited to, VTScada driver tags.

Users should wait until \SocketServerManager\Started is true before attempting to use the Register or Unregister functions.

GroupName defines a group containing driver tags and server socket tags. Typically the name of an IP Network Listener. Groups are for use by advanced VTScada programmers, allowing them to associate multiple IP Network Listeners.

Context provides the context for the discriminator calls. The StationKey must be unique within the StationList of a GroupNode\DiscrimNode.

The Station context must have Discriminator() and Connect () callbacks. The Discriminator module takes a sample of incoming data and returns a station key if it can understand the protocol. The Connect module will be called on the exact station context as indicated by the station key. Connect must return (a) the workstation name of the station server and (b) the object value of a context implementing an ExternalSocketConnect callback. This was intended for, but is not limited to, TCP/IP and UDP/IP port tags.

The service will not Connect() to a station unless the IP address of the remote stream passes an optional IPAllowString filter check. Utility functions ArrayToString() and StringToArray are provided in the SocketServerManager to ensure that the addresses are provided in the correct format, but in general, you can rely on the result returned from a pIPAddressList() widget, used in the tag configuration.

A station must be UnRegistered when about to change the StationKey or GroupName. The IPAllowString can be updated by calling register again.

Example:

```
ActiveDevAddr{PLC address to use in reads & writes
};
ListenerGroup{SocketServerManager group to join
};
IPAddressAllow{ Semicolon delimited string of IP addresses (or ranges
                        which can connect to this station};
  If Valid(Name) && \SocketServerManager\Started && Watch(1, Act-
iveDevAddr, ListenerGroup, IPAddressAllow);
  [
    CriticalSection(
```

```
      IfThen(Valid(ActiveDevAddr) != Valid(RegDevAddr) || Act-
iveDevAddr != RegDevAddr ||
            Valid(ListenerGroup) != Valid(RegGroup)   || Listen-
erGroup != RegGroup,
        \SocketServerManager\Unregister(Root, RegDevAddr, RegGroup);
      );
      RegDevAddr = ActiveDevAddr;
      RegGroup   = ListenerGroup;

       { Register with SocketServerManager }
      \SocketServerManager\Register(Root, RegDevAddr, RegGroup, IPAd-
dressAllow);
    );
  ]
```

See Also:

... SocketServerManager\UnRegister

SocketServerManager\StringToArray

Description Utility function to expand a semicolon-delimited string of IP addresses into an array of individual strings

Returns Array of text

Usage Script

Function Groups Stream and Socket

Format \SocketServerManager\StringToArray(AddressString)

Parameters

AddressString

Required. A text string of IP addresses, where each address is delimited from the next by a semicolon.

Comments None.

See Also:

...SocketServerManager\ArrayToString

SocketServerManager\UnRegister

Description: Unregister a station from a group. The station must be

unregistered whenever GroupName, or StationKey changes.

Returns: Invalid when complete

Usage: Script

Function Groups: Stream and Socket

Format: \SocketServerManager\UnRegister(Context, StationKey, GroupName)

Parameters:

Context

Required. The context should be the root tag. This must have Discriminator() and Context() subroutines.

StationKey

Returned by Discriminator() in the root tag. An opaque string used to identify a particular station.

GroupName

The name of an IPListener tag (or group). This will be the source of the new data streams.

Comments: Users should wait until \SocketServerManager\Started is true before attempting to use the Register or Unregister functions.

Example:

```
ActiveDevAddr{PLC address to use in reads & writes
};
ListenerGroup{SocketServerManager group to join
};
IPAddressAllow{ Semicolon delimited string of IP addresses (or ranges
                        which can connect to this station};
  If Valid(Name) && \SocketServerManager\Started && Watch(1, Act-
iveDevAddr, ListenerGroup, IPAddressAllow);
  [
    CriticalSection(
      IfThen(Valid(ActiveDevAddr) != Valid(RegDevAddr) || Act-
iveDevAddr != RegDevAddr ||
             Valid(ListenerGroup) != Valid(RegGroup)   || Listen-
erGroup != RegGroup,
        \SocketServerManager\Unregister(Root, RegDevAddr, RegGroup);
```

```
      );
      RegDevAddr = ActiveDevAddr;
      RegGroup   = ListenerGroup;

       { Register with SocketServerManager }
      \SocketServerManager\Register(Root, RegDevAddr, RegGroup, IPAd-
dressAllow);
    );
  ]
```

See Also:

... SocketServerManager\Register

Time Synchronization Manager Service

The Time Synchronization Manager Service synchronizes client clocks with the configuration server's clock.

Method

The Time Synchronization Manager Priority Service synchronizes time for a VTScada application as soon as the application is started. Clients then independently request time updates from the server both on startup and periodically (as defined by the TimeSyncUpdtItrvl application property). The server responds to these requests as they are received.
If synchronization with a centralized provider is required, configure the server to do so using the Windows® configuration tools.
The client transmits 5 update requests (in the form of GetTime calls) through RPC to the server. The server responds to each GetTime call with a SetTime RPC call that includes the server's current time in UTC. The client calculates the round trip time for each request/response, and will select the most expedient (i.e. the shortest request/response time).
After the client chooses the shortest round trip response time, the client's time delta from the server is calculated, taking the client's timezone into consideration.

Delta Tolerance

The delta tolerance is equal to the round trip time, with 0.5 seconds as the minimum delta. The synchronization interval is 15 minutes (900 seconds), as defined by the TimeSyncUpdtItrvl application property.
If the time delta is 1 second, then the client's clock is slewed by 10 milliseconds every second. The client will then be in sync after 100 seconds, but any logs done on time won't show any visible discontinuity. The client's clock will immediately be adjusted if the delta exceeds 5 seconds.

Related Information:

...Special Considerations for Time Adjustments

...Time Synchronization Manager Properties - See: The VTScada Admin Guide

Special Considerations for Time Adjustments

The following scenarios will cause the Time Synchronization Manager to abort a time adjustment to a client:

- If the adjusting the client's clock would result in rolling the client's data back to a previous day, the time adjustment will not occur, as doing so would have a negative effect on data logging, especially at the first day of a month, as the rollback would then be to the previous month.
- The client's clock will also not be adjusted if the delta from the server is less than some tolerance (relative to the roundtrip time of the request). For example, if a roundtrip takes a whole second, the server's timestamp will not be very precise, so the client's clock will not be adjusted unless the delta is > 0.5 seconds.
- The client's clock will not be adjusted if the roundtrip time exceeds 10 minutes.
- The client's clock will not be adjusted if the server's RPC queue (which takes priority) exceeds a user-defined value (as defined by the TimeSyncRPCQMax application property).
- No time synchronization will take place if the TimeSyncEnable application property is set to 0 (disabled).

Web Services and XML

The Web Services Module provides a means for VTScada to receive Simple Object Access Protocol (SOAP) requests over a network and translate those requests into VTScada procedural requests. The results of these requests can be compiled back into the SOAP message format and sent back to the remote computer.

Note: The ability to provide Web Services is a separate licensing option of VTScada. If you did not purchase a license for the Web Services module, it will not be available on your system.

This technology enables remote applications to make use of selected VTScada services that you choose to provide from your application. Possible applications range from remote reporting to complex delivery scheduling systems, relying on calculations done in VTScada to predict when material deliveries will be required.

The WebService module links together a WSDL document, a VTScada Realm, and a set of project-defined VTScada modules. This linkage enables VTScada to accept SOAP messages directed at specific VTScada Realms via the network and translate those messages into procedural requests for VTScada applications. The results of these requests can then be compiled back into SOAP message format and returned to the network entities that made the requests.
Realms can be accessed only by authorized users. Security must be enabled and there must be at least one account with the Internet Client Access privilege.
Each VTScada Realm can expose a single WSDL and connect it to a specific module within a single application. All VTScada modules to be called via that Realm must be direct submodules of the connected module, so that the WebService functions can locate them.
Subsequent WSDL attachments to the same Realm will supersede existing attachments, not aggregate with them.

Once a Realm is attached to a WSDL it can start accepting SOAP messages immediately and will continue to accept them until the attachment is removed. The individual calls can accept up to 30 parameters each. The parameters must be provided as defined by the WSDL and passed via the SOAP message. In the case that a complex type is sent as a parameter in the message, that parameter will be passed to the module as a XMLNode Tree.

Related Information:

...Terms Used with Web Services - Reference

...Web Services Process - How connections are made and information transferred.

...Module and Parameter Naming - Guidelines

...VTScada Web Service Commands - Functions to connect to a WSDL file.

...Web Services Example - Shows how to configure VTScada and the code (PHP in this example) to read information.

...VTScada Engine XML API - Full function reference.

Terms Used with Web Services

Term	Definition
DOM	Document Object Model. A W3C standard that defines a programmatic interface to a parsed XML document. The DOM presents an easily processed, standardized interpretation of an XML document.
DTD	Data Type Definition. A declaration and optional syntactic rules that an XML document must adhere to. DTDs are still legal, but have largely been superseded by XML schema.
RPC	Remote Procedure Call.
SOAP	SOAP (originally Simple Object Access Protocol) is a protocol for exchanging XML-based messages over computer

	networks, normally using HTTP.
SOM	Schema Object Model. A Microsoft implemented API that provides a procedural API to schema cached in their XML engine.
W3C	World-Wide Web Consortium. An organization that maintains the standards used on the Internet, including the specifications that relate to XML and XML schema.
Web Service	The W3C defines a Web service as a software system designed to support inter-operable Machine to Machine interaction over a network. Web services are often just Web APIs that can be accessed over a network, such as the Internet, and executed on a remote system hosting the requested services.
WSDL	Web Services Description Language is an XML-based language that provides a model for describing Web services. The World Wide Web Consortium provides a description of WSDL and a set of examples for using it at http//www.w3.org/TR/wsdl
XML	eXtensible Markup Language. A W3C standard for the representation of arbitrary data and associated properties using a markup language.
XML Document	An XML stream that is complete, having a valid XML declaration and a complete set of balanced tags.
XML Processor	A VTScada engine internal entity that exposes a script code interface to allow an XML document to be represented in a manner easy for script to access and manipulate. The XML processor is capable of parsing and writing XML documents.
XML Schema	An XML document that specifies additional syntactic rules that an XML document that "conforms" to the schema must adhere to.
XSLT	XML Style Sheet Transformation. An XML document that

contains a set of transforming instructions. Such a transform can be used by an XML processor to generate a new W3C standards-compliant output document from an XML document, e.g. (X)HTML output.

Web Services Process

The WebService module works by linking together a Web Services Description Language (WSDL) document, a VTScada Realm, and a module within a VTScada application.

This linkage enables VTScada to accept SOAP-encoded messages, which are directed at specific VTScada Realms, and to translate those messages into procedural requests for VTScada applications. The results of these requests can be compiled back into SOAP message format and returned.

Each VTScada Realm can expose a single WSDL and connect it to a one specific module within a single application.

The module commonly takes the form of a .SRC file in the application's main directory. It is reserved by the system as the root of the web service. Sub-modules of this root are used to expose parameters and VTScada operations to the WebService.

Only one module can be associated with a realm, but since this module may contain submodules that are available to the web service, an effectively unlimited number of SOAP calls can be handled via the single root module

Do not confuse the module with the application pages. None, any or all tags from any or all application pages may be exposed to the web service through the module system.

A key feature of the module will be a call to the WebService function, SetWSDL. This function connects a Realm to the application in order to provide the web service interface. An example of a call to this function can be found in the Web Services Example: Creating the VTScada Module.

The WSDL file describes the structure of the web service. At a minimum the WSDL file will define, in XML format:

- The operations that are available.
- The data types of variables passed and returned.
- The location of the Realm to which the application is linked.

The WSDL file must also include the name of the module to be called. Incoming SOAP messages will use this same name to indicate the module to call. Outgoing messages will use the name of the module with the string "Response" appended, thereby indicating their source.

Note: VTScada is compliant with SOAP 1.1 not with SOAP 1.2. Messages must be encoded for SOAP 1.1.

Once the application has been configured it can start accepting SOAP messages immediately and will continue to accept them until the attachment between the realm and the WSDL is removed.

Related Information:

...Terms Used with Web Services - Reference

...Module and Parameter Naming - Guidelines

...VTScada Web Service Commands - Functions to connect to a WSDL file.

...Web Services Example - Shows how to configure VTScada and the code (PHP in this example) to read information.

...VTScada Engine XML API - Full function reference.

Module and Parameter Naming

The WebService functions use name matching on the WSDL[1] <operation> tags in order to discover SOAP[2] modules. Modules to be called must therefore be named such that they match the names of the <operation ...> tags described in the WSDL document. Incoming SOAP messages will use this same name to indicate the module to call, while outgoing messages will use the name of the module with the string "Response" appended to indicate their source.
e.g.

```
    <wsdl:message name="GetTagValueInput">
        <wsdl:part name="request" type="tns:GetTagValueIn" />
    </wsdl:message>

    <wsdl:message name="GetTagValueOutput">
        <wsdl:part name="response" type="tns:GetTagValueOut" />
    </wsdl:message>
    <wsdl:portType name="TagQueryServicesPort">
        <wsdl:operation name="GetTagValue"
                        parameterOrder="request response">
           <wsdl:input message="tns:GetTagValueInput"/>
           <wsdl:output message="tns:GetTagValueOutput"/>
        </wsdl:operation>
    </wsdl:portType>
```

The VTScada module referenced here, GetTagValue, would have a structure similar to the following:

```
<
GetTagValue
(
  request;
  response;
)
...
>
```

Parameters exposed by the called modules must cover both input from the incoming SOAP message and output to the outgoing SOAP response.

---

[1]Web Services Description Language. An XML-based language that provides a model for describing Web services
[2]SOAP (originally Simple Object Access Protocol) is a protocol for exchanging XML-based messages over computer networks

The names of these parameters must match the <part> tag names associated with both the <input> tags and <output> tags as defined in the <operation> section of the WSDL for the called module.
This system enables parameters to be properly integrated into messages. Each output parameter will be given a pointer to a blank instance of the type required by the method. The method should fill out these structures, extending array portions as necessary. Similarly, each input parameter will be passed a pointer to the incoming data type that the parameter represents. The pointers themselves must not be changed as this would break the linkage to the WebService system calling the module and prevent proper passage of output data.
The return value of a called module must not be of type OBJECT and should be INVALID. Steady-state called modules must not contain a return statement at all as this will cause VTScada to attempt to execute them as subroutines (i.e. procedurally). Any values that are returned will simply be discarded by the system.
Once a web service is set up, a variable called "WSDrvr" will be added to the connected module

Note: Developers should not include a variable with the name "WSDrvr" themselves. The WebService initialization function will reject any modules containing such a variable.

This variable records several pieces of context data relevant to the new web service as well as a group of support functions for common web service processing tasks.

VTScada Web Service Commands

The WebService module exposes two methods for use during web service deployment, SetWSDL and RemWSDL. The first performs the attachment of a WSDL to a Realm and the second disconnects such an attachment if it exists.

SetWSDL connects a Realm with a WSDL file and a set of VTScada modules in order to enable a web service interface. Linkage is first applied between the WSDL file and the VTScada modules by generating an XML schema using the WSDL and the parameters provided to this function. The Realm's address is then registered with the VTScada HTTP server to connect the whole thing to the network.
A prototype for a call to the SetWSDL function follows. It will return 0 for success and 1 for failure. If an error occurs, a message will be returned via the pointer, pResponse.

```
\System\WebService\SetWSDL(<WSDL File Path>,
                           <Realm>,
                           <Call Scope>,
                           <Service>,
                           <pResponse>);
```

RemWSDL disconnects a Realm from a WSDL file and the associated set of VTScada modules. It then proceeds to clean up any resources consumed by the web service that this association represented. After RemWSDL is called, the associated web service will immediately stop processing messages. Note, however, that any operations set in motion by that service will run to completion. This function is called implicitly if the connected module is destroyed.
A prototype for a call to the RemWSDL function:

```
\System\WebService\RemWSDL(<Realm>);
RemWSDL must be called in a script.
```

Related Functions:

... SetWSDL

... RemWSDL

WSDrvr Services

Once a web service is successfully registered via SetWSDL, a variable named "WSDrvr" is added to the connected module (see: VTScada Web Service Commands). This variable exposes a set of data points for use by

the web service and the WebService functions, as well as a small group of helper functions. The data points are presented to the web service on a READ ONLY basis – developers should not attempt to modify these values as it will adversely affect their web service.
The Web Service Helper Functions are described following the data points:

Web Service Data Points

\WSDrvr\Realm	This is a copy of the realm name to which the web service is attached.
\WSDrvr\Rscope	This is an object reference to the connected module.
\WSDrvr\WSDL	A copy of the WSDL in use. This has not been converted into VTScada format; it is a raw text dump.
\WSDrvr\XMLHandle	The XML processor used for this web service, it contains the basic schemas for SOAP processing as well as a schema-converted version of the WSDL.
\WSDrvr\MsgNamespace	The target namespace of the WSDL, used by all messages and operations.
\WSDrvr\XMLns	A list of "xmlns" declarations delineating all of the namespaces used in the WSDL and basic SOAP schemas (and therefore all namespaces usable by this web service). Any messages generated by this service must have at least a subset of these declarations attached.
\WSDrvr\NSminus	Due to a rule of SOAP 1.1, the tags in a response message to a SOAP request should not (and in most cases must not) have namespace prefixes if they are declared in the "MsgNamespace". This is a list of all other namespace / prefix pairs declared for the service and is used to generate output messages.
\WSDrvr\Nsp	This object enables rapid conversion of namespaces to prefixes. It consists of a group of variables with

	names matching the namespaces declared for this web service (note that these contain "illegal" variable name characters, and can only be referenced via the Scope keyword), the value of each being a string containing the prefix associated with that namespace.
\WSDrvr\Pfx	This object enables rapid conversion of prefixes to namespaces. It consists of a group of variables with names matching the namespace prefixes declared for this web service, the value of each being a string containing the namespace name itself.
\WSDrvr\CallIdx	An object containing linkage data between messages and modules. This is of use only to the WebService functions.
\WSDrvr\ClassFactory	The XML Schema Cache Dictionary object used by the Web Service connected to the given module, it can be used to instantiate any of the types in any of the schemas registered by this service. The user of this service is free to add additional schemas to the Schema Cache Dictionary object as desired. Each Web Service has its own ClassFactory object in order to prevent a single damaged service from affecting others.

Web Service Helper Functions

\WSDrvr\MakeTypeArray

Description:	The MakeTypeArray function creates a SOAP 1.1 compliant array within the parent XMLNode provided and of the size specified.
Returns:	Invalid.
Usage:	Script
Format:	\WSDrvr\MakeTypeArray(pParent, Namespace, Name, Size, MemberNamespace, MemberName);

Parameters:

pParent

A pointer to the XMLNode where the array will be added.

Namespace

The namespace of the array type.

Name

The name of the array.

Size

The size of the array.

MemberNamespace

The namespace of the member type.

MemberName

The name of the member type

Comments: The function creates an array of the specified size under a member called Name in the parent XMLNode. If the MemberName can be found in the schema cache under the MemberNamespace, then each element of the array will be of the specified type. Otherwise, it will be a plain XMLNode.

\WSDrvr\MakeTypeInstance

Description: The MakeTypeInstance function creates a copy of an XMLNode representing an XMLType.

Return Value: The XMLNode representing the type or invalid if the type cannot be found.

Usage: Script

Format: \WSDrvr\MakeTypeInstance(Namespace, Name)

Parameters:

Namespace

The namespace of the type

Name

The name of the type

Comments: This function creates a clone of the XMLType specified by Name in the Namespace. If the type doesn't exist in the Namespace or the Namespace doesn't exist in the Schema Namespace Cache (ClassFactory) then invalid is returned.

\WSDrvr\ReportFault

Description: Used to raise an engine level SOAP fault from within the service, causing the next processing phase to abort and a SOAP fault packet to be returned to the client.

Return Value: Invalid

Usage: Script

Format: \WSDrvr\ReportFault(Description, Detail, ProcessFault);

Parameters:

Description

A string describing the fault.

Detail

A string providing extra detail information.

ProcessFault

0 indicates an input problem, 1 indicates a processing problem.

Comments: The SOAP fault can be attributed to either local web service processing or poor input data from the client. Detailed fault information must be added to the message.

\WSDrvr\GetAttValue

Description: GetAttValue returns the value of the specified XML attribute (stored as value metadata) within a particular tag representation.

Usage: Script

Return Value: The value of the attribute if found, otherwise Invalid

Format: \WSDrvr\GetAttValue(pTag, Name, NameSpace);

Parameters:

pTag

A pointer to the variable to be inspected.

Name

The name of the attribute to be retrieved.

NameSpace

The namespace of the member type.

Comments: The tag representation may functionally be any variable. The name of the attribute to retrieve is first attempted to be resolved as a full QName, and if no match is found then just the name of the attribute is resolved. QName construction is first attempted by using the NameSpace parameter, and then by getting the namespace of the tag.
A QName is a qualified name. For example, <tns:myElement tns:myAttrib="abc">. In this case "tns:myAttrib" is the QName. Thus, if a Namespace is provided, then the prefix for that namespace is looked up and preprended to the name and then tried first when searching for an attribute. If not found, then the plain Name is used.

Related Functions:

... SetWSDL

Web Services Example

A relatively straight-forward example of a web service is to use a CGI application running on a remote website to read tag values from a VTScada application.
The following are minimum requirements:

- A network connection.
- A running application that includes one or more tags with values to be read.
- Security is enabled and at least one account possesses the Internet Client Access Privilege.
- A named realm that makes the application available.
- A WSDL file in the application directory that describes the service.
- A module in the application that provides the web service
- (Optional, but useful to ensure that the service is instantiated at startup:) A line in the application's AppRoot.src module to name the service module above.
- A program running remotely to call and use the web service.

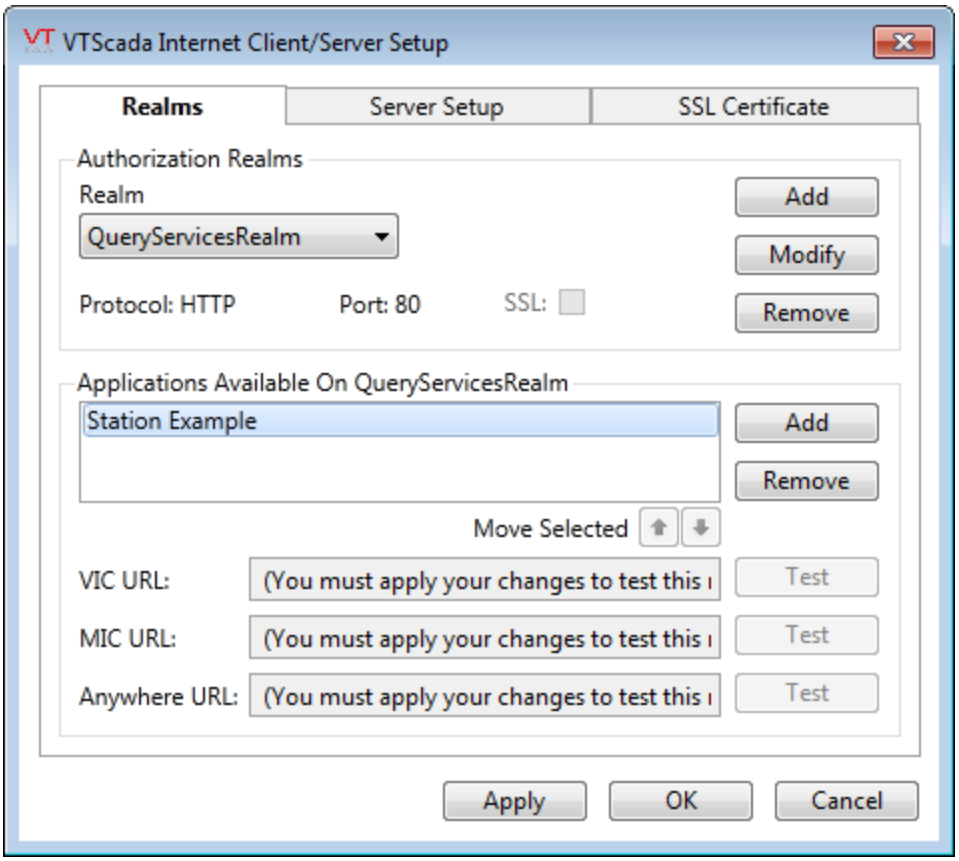
Assuming the first two items, the network connection and a working application are in place, the following steps will describe the process of setting up a web service to read values from the application from a PHP-enabled web page.
This example uses an application named StationExample which is located in the directory C:\VTScada\StationExample. It contains a single analog input named AI20_1.

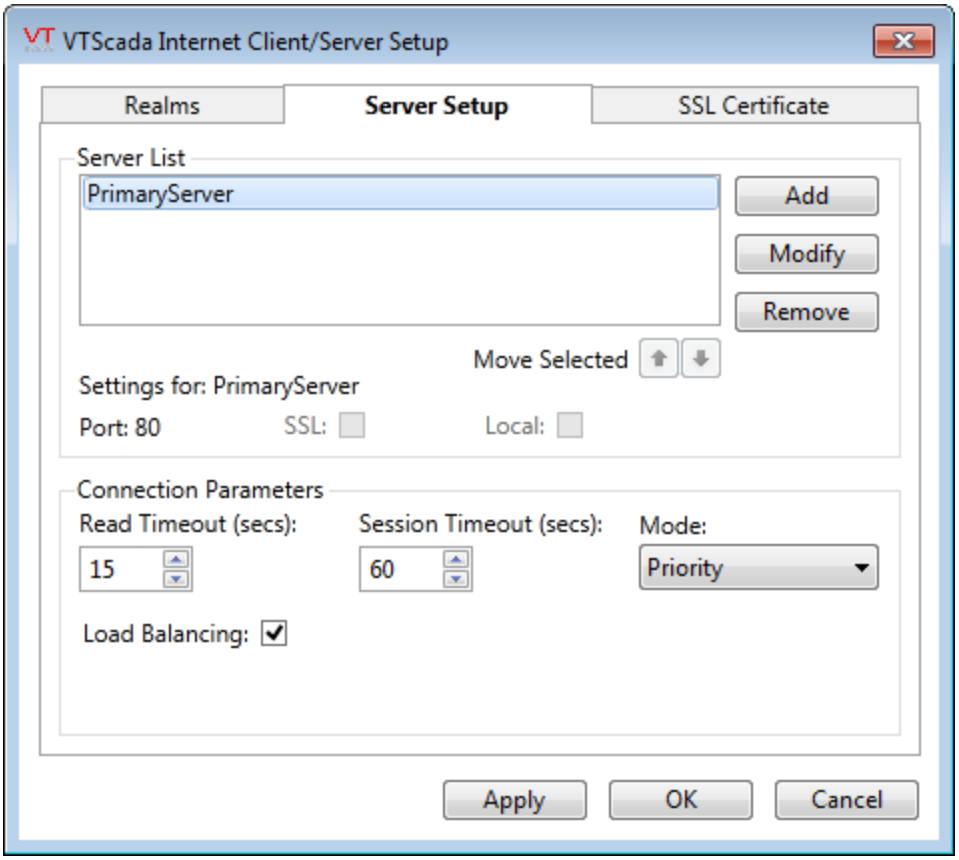
Next Steps:

...Configuring a Realm

Configuring a Realm

The Realm provides the gateway between a VTScada application and the internet. For this example, the realm will be named QueryServicesRealm and will point to a single application: Station Example.
One must also configure VTScada to be a server via the Server Setup tab (as discussed in Browsing VTScada Applications On The Web). The completed dialog box is shown in the following two figures:


VTScada Internet Client/Server Setup
Realms
Server Setup
SSL Certificate
Authorization Realms
Realm
QueryServicesRealm
Add
Modify
Remove
Protocol: HTTP
Port: 80
SSL:
Applications Available On QueryServicesRealm
Station Example
Add
Remove
Move Selected
VIC URL:
(You must apply your changes to test this
Test
MIC URL:
(You must apply your changes to test this
Test
Anywhere URL:
(You must apply your changes to test this
Test
Apply
OK
Cancel


VTScada Internet Client/Server Setup
Realms
Server Setup
SSL Certificate
Server List
PrimaryServer
Add
Modify
Remove
Move Selected
Settings for: PrimaryServer
Port: 80
SSL:
Local:
Connection Parameters
Read Timeout (secs):
15
Session Timeout (secs):
60
Mode:
Priority
Load Balancing:
Apply
OK
Cancel

Next Steps:

...Creating a WSDL File

Creating a WSDL File

The Web Services Description Language file provides the model for describing the web service.
The file commonly starts with a header which includes links to the XML schema. To adapt the following header to your own system, replace every entry that reads "localhost" with the fully qualified domain name of your own server. ('localhost' will probably work on your system for testing purposes)

Note: XML requires that the target Namespace be a globally unique identifier. Common practice would be to use a fully qualified domain name here. "localhost" only works for our example because the application is running on the same computer as the "remote" CGI script.

```
<?xml version="1.0"?>
<wsdl:definitions name="TagQueryServices"
   targetNamespace="http://localhost/"
   xmlns:tns="http://localhost/"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
   xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
   xmlns:s="http://www.w3.org/2001/XMLSchema">
```

After the header comes a section describing the WSDL types. This describes the target namespace (again, localhost is used here - substitute your own domain), and the expected input and output variables. In this case, when a request is made for 'TagName' a response will be generated that includes 'TagValue' and a 'ReturnCode'. The TagName is expected to be of type String. The TagValue will be of type Float and the ReturnCode will be an integer.

```
<wsdl:types>
  <s:schema
      targetNamespace="http://localhost/">
    <s:import
      namespace="http://schemas.xmlsoap.org/soap/encoding/" />
    <s:import
      namespace="http://schemas.xmlsoap.org/wsdl/" />
    <s:complexType name="GetTagValueIn">
      <s:sequence>
```

```
        <s:element
          minOccurs="1"
          maxOccurs="1"
          name="TagName"
          type="s:string" />
      </s:sequence>
    </s:complexType>
    <s:complexType name="GetTagValueOut">
      <s:sequence>
        <s:element
          minOccurs="1"
          maxOccurs="1"
          name="TagValue"
          type="s:float" />
        <s:element
          minOccurs="1"
          maxOccurs="1"
          name="ReturnCode"
          type="s:int" />
      </s:sequence>
    </s:complexType>
  </s:schema>
</wsdl:types>
```

The expected SOAP messages are defined next: In this case there are only two: GetTagValueInput (the request) and GetTagValueOutput (the response).

```
<wsdl:message name="GetTagValueInput">
  <wsdl:part name="request" type="tns:GetTagValueIn" />
</wsdl:message>
<wsdl:message name="GetTagValueOutput">
  <wsdl:part name="response" type="tns:GetTagValueOut" />
</wsdl:message>
```

The PortType section includes a supported set of operations, in this case "GetTagValue".
Each operation lists the input and the output messages of the operation.

```
<wsdl:portType name="QueryServicesPort">
  <wsdl:operation
    name="GetTagValue"
    parameterOrder="request response">
    <wsdl:input message="tns:GetTagValueInput"/>
    <wsdl:output message="tns:GetTagValueOutput"/>
  </wsdl:operation>
</wsdl:portType>
```

The Binding section ties the SOAP calls to the supported operations. Note the use of the 'localhost' address in this section. Again, you will need to change this to the domain of your own server. Also to be noted is the reference to the port defined above.

```
<wsdl:binding
  name="QueryServicesSoapBinding"
  type="tns:QueryServicesPort">
  <soap:binding
    style="document"
    transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="GetTagValue">
    <soap:operation soapAction="http://localhost/GetTagValue" style-
e="rpc"/>
    <wsdl:input>
      <soap:body
        use="encoded"
        namespace="http://localhost/"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
      </wsdl:input>
    <wsdl:output>
      <soap:body
        use="encoded"
        namespace="http://localhost/"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
    </wsdl:output>
  </wsdl:operation>
</wsdl:binding>
```

At the end of the WSDL file is the most interesting part. This small block of code ties the operations defined above to the actual realm that will be called from our external program. Note the name – this will appear again in the next step which creates the VTScada module to handle the requests. Also note the appearance of the realm name after the domain when defining the address for the SOAP requests.

```
<wsdl:service name="TagQueryServices">
<wsdl:port
name="QueryServicesPort"
binding="tns:QueryServicesSoapBinding">
<soap:address
location="http://localhost/QueryServicesRealm/"/>
</wsdl:port>
</wsdl:service>
```

To finish the file, close the tag that opened it all:

```
</wsdl:definitions>
```

Next Steps:

...Create the VTScada Module

Create the VTScada Module

To fully understand the following block of code requires some knowledge of VTScada's programming language. However, the important points are easily described:
GetTagValue is the name of the subroutine which will do exactly that function: get the requested tag's value.
The ServiceActive line includes a call to the function SetWSDL. This function requires four parameters:

- The location of the WSDL file,
- The name of the realm,
- The scope of the call (in this case, 'self' – the current module),
- The name of the service – which was pointed out in the preceding section on the WSDL file

The 5th parameter is ErrMsg: a pointer to an error message to return should SetWSDL fail. While optional, this parameter is strongly recommended as it can be invaluable when debugging.
The MAIN section of the code is somewhat cryptic, but what it does is rather simple: It takes any given tag name and attempts to find the current value associated with that tag. If found, the value will be returned along with a "0" to indicate success. Otherwise, a value of "-1" is returned to indicate failure.
Text lines between braces {...} are comments.

```
{========================== TagQueryServices
=========================}
{ This module contains the code that implements the services
provided. }
{=====================================================================-
==}
[
  ServiceActive { Return value from SetWSDL                    };
  GetTagValue Module;
  ErrMsg;       { Error message from SetWSDL. Invalid if none. }
]
Init [
  If Watch(1);
  [
    ServiceActive = \System\WebService\SetWSDL(
                     "file://C:/vts/Station1/TagQueryServices.wsdl",
                     "QueryServicesRealm",
                     Self,"TagQueryServices",
```

```
                      &ErrMsg);
  ]
]
<
{============================ GetTagValue
==========================}
{ Subroutine called through SOAP to request the value of a tag. }
{=========================================================================-
=}
GetTagValue
(
  request;
  response;
)
[
  TagName;
  TagObj;
]
Main [
  If 1;
  [
    TagName = (*request)\TagName;
    TagObj = Scope(\Code, TagName);
    IfElse(Valid(TagObj), Execute(
      (*response)\TagValue = TagObj\Value; { get the tag's value }
      (*response)\ReturnCode = 0; { success }
    );
    {Else}
      (*response)\ReturnCode = -1; { error - tag doesn't exist }
    );
    Return (0);
  ]
]
>
```

Next Steps:

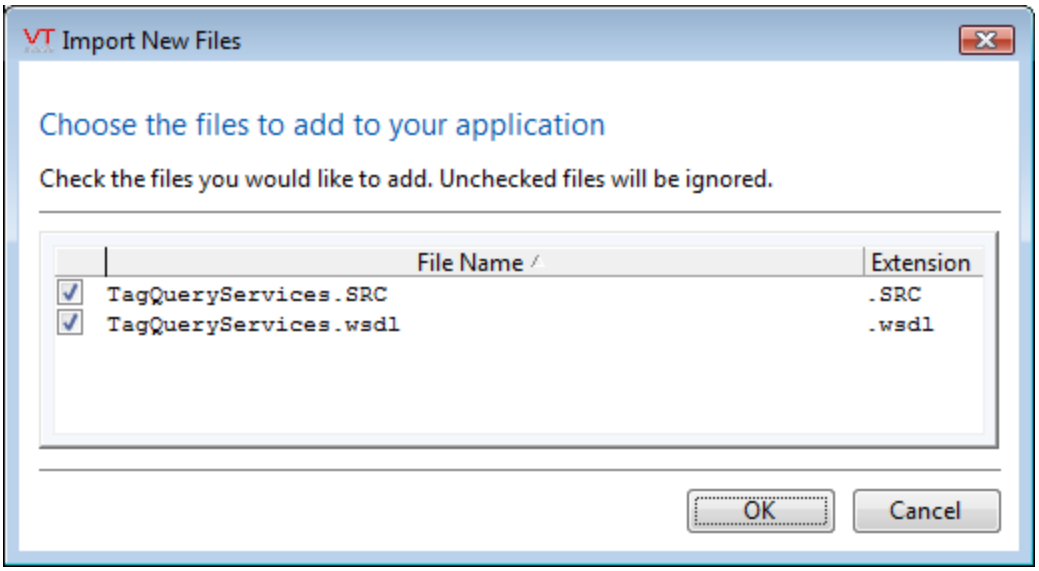
...Modifying AppRoot.SRC

Modifying AppRoot.SRC

The final step on the VTScada side of supplying web services is to add a line to the Services section of the application's AppRoot.SRC file to tell it where to find the module created in the last step. This step is required only if you want the web service to be instantiated at start up.
For this example, it will be declared as:

```
[ (SERVICES) {=== Modules that are services that are started ===}
  TagQueryServices Module "TagQueryServices.SRC";
]
```

Again, note the name "TagQueryServices" that was defined in the WSDL file and then used again within the module TagQueryServices.SRC.
After adding this line, the application will need to be re-compiled. Stop it if it is running and run the Compile command from the VAM. You will be prompted to import the new files:



When the application starts again it will be providing a web service of returning requested tag values from our application.

Next Steps:

...Requesting Values via the Web Service

Requesting Values via the Web Service

This example uses a PHP web page to interact with the VTScada web service.

Note: The point of this example is to show that any program that can access web services can now interact with VTScada. You could call the web service from another VTScada application, from a CGI script in a web page as shown here, or from your own web-enabled application.

While the following code does work, it should not be taken as representing a fully developed application. PHP was selected for this example as it is freely available and relatively easy to use, but problems

that can be traced to the PHP configuration have been reported.

Success when using web services will require that you understand the SOAP protocol support in your chosen client.

** Trihedral Engineering provides no support for programs other than VTScada. **

The process of making the call to VTScada's web services is to send a SOAP-encoded message, calling the operations that were exposed with the WSDL file. Depending on your application, you may or may not need to include a header with the SOAP-encoded message. If so, the header would look something like this:

```
POST /TagQueryServicesRealm/SOAP/ HTTP/1.1
Host: localhost
Content-Type: text/xml; charset="utf-8"
Content-Length: 343
SOAPAction: "http://localhost/GetTagValue"
```

In the case of this PHP example(*), this header is not required as the PHP SOAP client object takes care of it automatically.

```
<html><head><title>VTScada Web Services Tester</title></head>
<body bgcolor="#ffffff">
<?php
  echo "<h2>VTScada Web Services Tester</h2>";
  // create an instance of PHP's SOAP client object
  // the client should not need a local copy of the .WSDL. It can
pull it from the server
  $client = @new SoapClient("http://-
localhost/QueryServicesRealm/WSDL");
  $params = array('TagName'=>'AI20_1');

  try {
    $result = $client->GetTagValue($params);
    $tagvalue = $result->TagValue;
    echo "The value of AI20_1 is ".$tagvalue;
  } catch (SoapFault $exception) {
    echo $exception;
  }
?>
</body>
</html>
```

Full tag names are supported. For example, to access a tag from the Completed Tutorial Example, use:

```
$params = array('TagName'=>'Local TCP Port\PLCSim\Pump 1\Motor
Speed');
```

If you are attempting to follow this example and errors are returned, you are advised to do a web search using the text of those error messages. A scan of online forums related to making SOAP calls from PHP will reveal a number of common problems and solutions. Messages in your PHP error log file may also be useful for this search. In particular, be careful to configure VTScada and Apache with differing port numbers, and if VTScada is using a number other than 80, adjust the addressing in the client to match.

If security is enabled in your VTScada application, then ensure that your account has the Internet Client Access privilege and modify the SoapClient call as follows:

```
  $client = @new SoapClient("http://-
localhost/QueryServicesRealm/WSDL",array('login' => "Your User Name",
'password' => "Your Password"));
```

For the sake of simplicity in the example, no effort is made to protect the user name and password. Keep security in mind as you develop your own client application.

VTScada Engine XML API

Note: Warning: Applications created prior to VTS 10 that used XML must be re-coded to be compatible with changes made to the API in that release.

The XML API is used to create an XML Processor, which is a script code interface to allow an XML document to be represented in a manner easy for script to access and manipulate. An XML Processor serves as a conduit between an XML document and an application that will do something with the content found in that document.

Related Information:

...Validating versus non-Validating XML Processors

...The Schema Cache Dictionary

...XMLNodes

...Accessing a portion of an XMLNode tree.

...Obtaining a list of child tags

...Determining if a member is an XMLNode or an array of nodes

...Assigning values to an array of XMLNodes

...Adding or deleting child tags

...XML Namespaces

Validating versus non-Validating XML Processors

The XML Processor can be either validating or non-validating, depending on whether it has an optional schema cache as specified in a module designated as a Schema Cache Dictionary. If it is created to have a schema cache, it is a validating processor.

A non-validating XML Processor checks the XML supplied to it for being "well-formed", meaning that the XML is syntactically correct.

A validating XML Processor goes further by checking that the structure of the supplied XML conforms to the structuring rules specified by the schemas. To do this, the XML Processor's schema cache must be given all the schemas referenced by the XML, and all the schemas referenced by the added schemas. (Note: the root XML schemas are excepted since these are already built-in to the XML Processor)

A validating XML Processor also creates type definitions for all structures described by the schemas. These type definitions are added to the Namespace Schema Cache.

When a non-validating XML Processor successfully parses the XML supplied, it generates an XMLNode Tree.

When a validating XML Processor successfully parses the XML supplied, it generates an XMLNode tree. Any type definitions encountered will be added to the Namespace Schema Cache.

Related Information:

...The Schema Cache Dictionary

...XMLNodes

For further information on XML Schemas, please refer to the following two resources:

...http://www.w3.org/2001/XMLSchema

...http://www.w3.org/2001/XMLSchema-instance

The Schema Cache Dictionary

This is a dictionary of namespaces, keyed by XML namespace. As "SchemaCacheDictionary" it is required by the XMLProcessor function, which creates an XML handle. The contents of a schema cache dictionary are XMLNode representations of the XML types.

The schema cache dictionary replaces the ClassFactory, which was used in versions of VTS prior to 10.0

Related Information:

...XMLNodes

XMLNodes

An XMLNode has 6 members known respectively as: #content, #attribs, #namespace, #control, #cdata and #comment. If an XML tag has child tags, then they are represented as additional members of the structure. For example, given the following XML code:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Simple XML to test for well-formedness -->
<catalog>
  <book id="book42">
    <author>Pomeroy, Steve</author>
    <title>RPC Manual</title>
    <genre>SCADA Software</genre>
  </book>
</catalog>
```

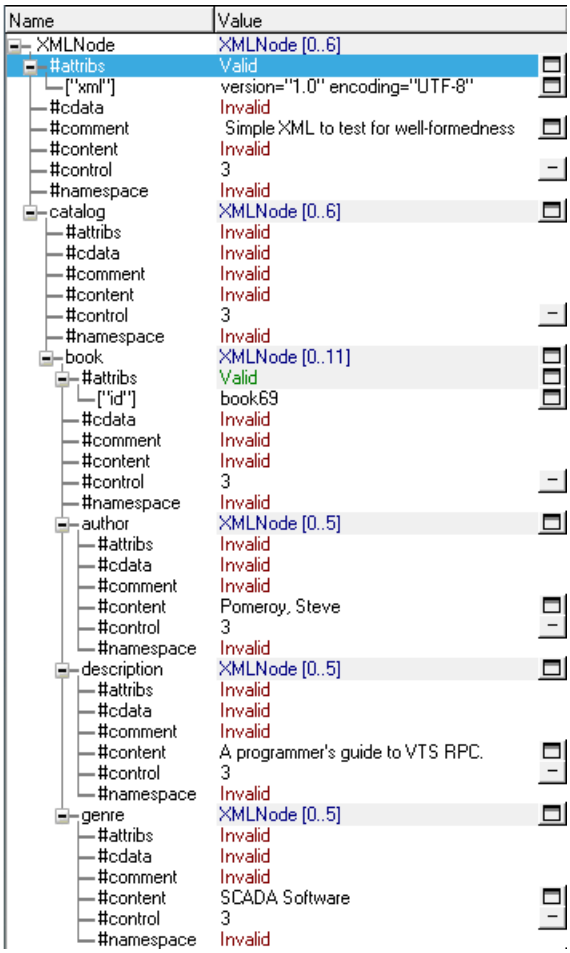
This would be represented in VTScada by the following XMLNode structure:

The top node represents the XML document. The attributes of this node correspond to the document's Processing Instructions.

Attributes are entries in a dictionary in the #attribs member.

The Debugger shows the nodes in alphabetic order. The #content member is actually at subscript 0, allowing script code to the #content member of any node just by specifying the node.

Tag contents are held in the #content member



The fact that the #content member is at subscript [0] enables script code to take advantage of automatic subscription into arrays, to refer to the #content member of any node just by specifying the node. For example, to access the book title, assuming the XMLNode tree is held in variable "XMLNodes," use the following code:

```
XMLNode\catalog\book\title
```

All other members can be accessed via Scope():

```
Scope(XMLNode\catalog\book, "#attribs")
```

Both these constructs can be used on either side of an assignment.
The purpose of the 6 standard members of the structure are as follows:

Member Name	Purpose

#content	The textual content of the tag.
#attribs	A dictionary containing the attributes of the tag.
#namespace	A string representing the namespace the tag belongs to.
#control	An internal use field. Used by XMLWrite to spot loops.
#cdata	A string representing any CDATA associated with the tag.
#comment	A string representing a comment associated with the tag.

All members can be both read from and written to, but note that the #control member will be altered by XMLWrite() when processing the XMLNode tree. As each node is visited, the #control member is set to a unique value for that call to XMLWrite(). If a node already has the unique value when visited then a loop has been found and writing terminates at that point. An error is returned.

Accessing a portion of an XMLNode tree.

Take care when writing code to access a portion of an XMLNode tree. You should use either the address-of operator (&) to pass a pointer, or use XMLGetNode() to extract the node of interest:

```
MySub(&(XMLNode\catalog\book));
```

or

```
MySub(XMLGetNode(XMLNode\catalog\book);
```

Note that the first version will require the use of the dereference (*) operator on every access.

Related Information:

...XMLNodes

Obtaining a list of child tags

When code requires a list of all the child tags in a tag use the ListKeys() function. This will also return the 'fixed' members of the structure. Child members can be found starting at subscript [6] if the keys are returned in creation order by setting the Order parameter of ListKeys() to 'TRUE'. Example:

```
Members = ListKeys(XMLNode\catalog\book, 1);
```

Related Information:

...XMLNodes

...Accessing a portion of an XMLNode tree.

...Determining if a member is an XMLNode or an array of nodes

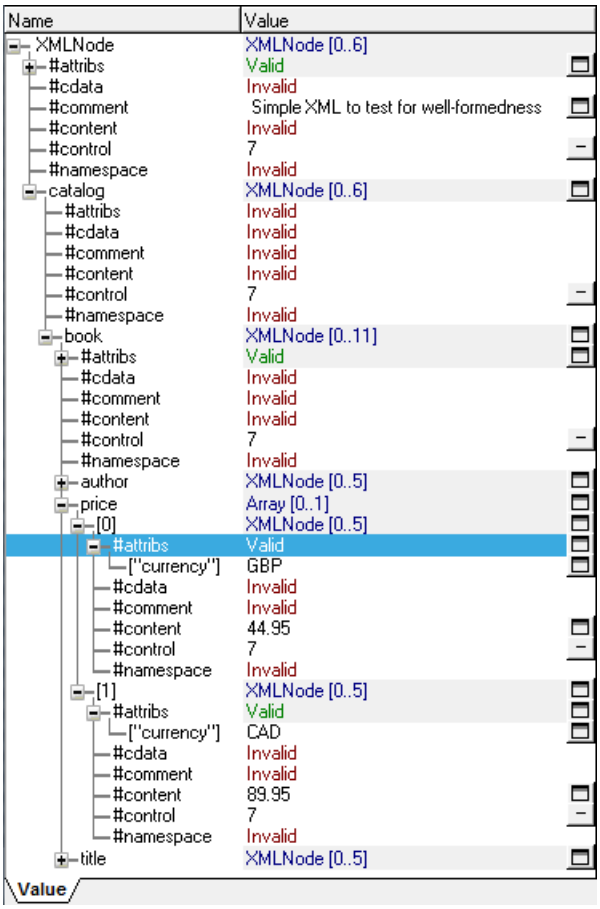
Related Functions:

... ListKeys

Determining if a member is an XMLNode or an array of nodes

When an XMLNode contains multiple instances of tags with the same name, then these are represented in the XMLNode tree by the member containing an array of XMLNodes. For example, given the following XML and its XMLNode representation of the price tag:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Simple XML to test for well-formedness -->
<catalog>
  <book id="book69">
    <author>Pomeroy, Steve</author>
    <title>RPC Manual</title>
    <price currency="GBP">44.95</price>
    <price currency="CAD">89.95</price>
  </book>
</catalog>
```



To determine if a member is an XMLNode use XMLGetNode():

```
Var1 = Valid(XMLGetNode(XMLNode\catalog\book));
Var2 = Valid(XMLGetNode(XMLNode\catalog\book\price));
```

Var1 will be set to 'TRUE', Var2 will be set to 'FALSE'.

Assigning values to an array of XMLNodes

When assigning values to an array of XMLNodes, such as the price array in the previous topic, neither ArrayOp1() or ArrayOp2() can be used. Instead a While or Do loop must be used.

Related Information:

... Determining if a member is an XMLNode or an array of nodes

Related Functions:

... WhileLoop

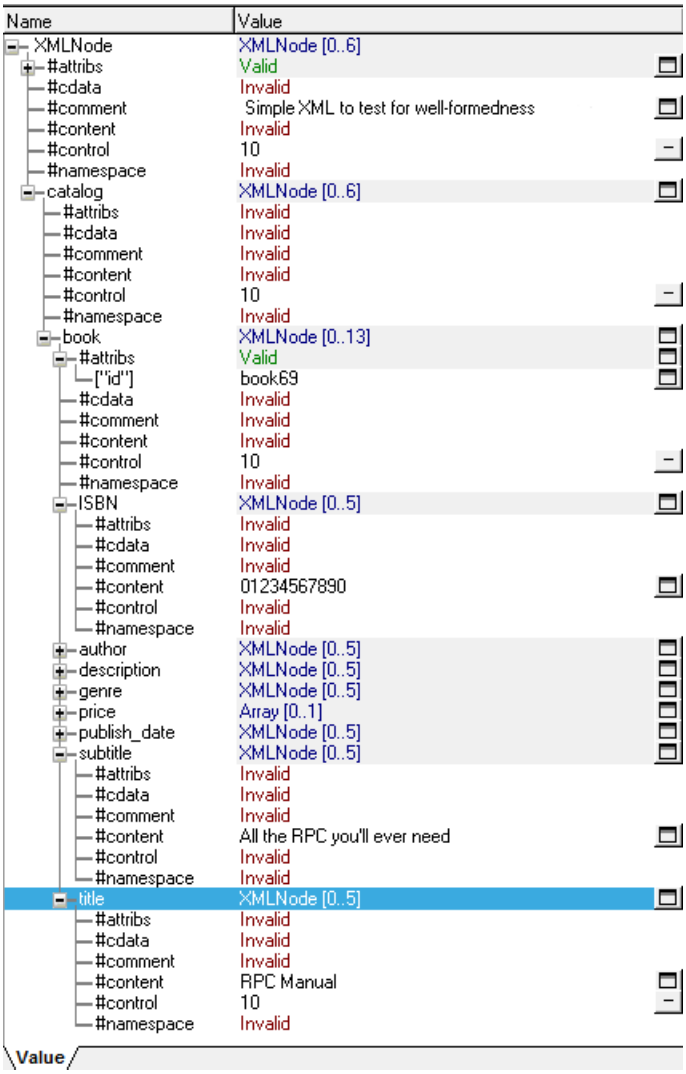
... DoLoop

Adding or deleting child tags

Members are added using XMLCloneNode() and removed with XMLDeleteMember(). XMLCloneNode() takes a dictionary of new members and their values and adds them to a copy of the specified node which is then returned as a result of the call. This enables addition of multiple members at the same time:

```
MembersDict = Dictionary(0);
MembersDict["ISBN"] = XMLCreateNode("01234567890");
MembersDict["SubTitle"] = XMLCreateNode("All the RPC you'll ever
need!");
XMLNode\catalog\book = XMLCloneNode(XML\catalog\book, MembersDict);
```

Results in the following XMLNode tree (building on the book example shown in earlier topics).



Deleting a member requires the use of XMLDeleteMember() specifying the XMLNode and the member, as shown:

```
XMLDeleteMember(XMLNode\catalog\book, "ISBN");
```

Note that the deletion is done 'in-place'.

Related Functions:

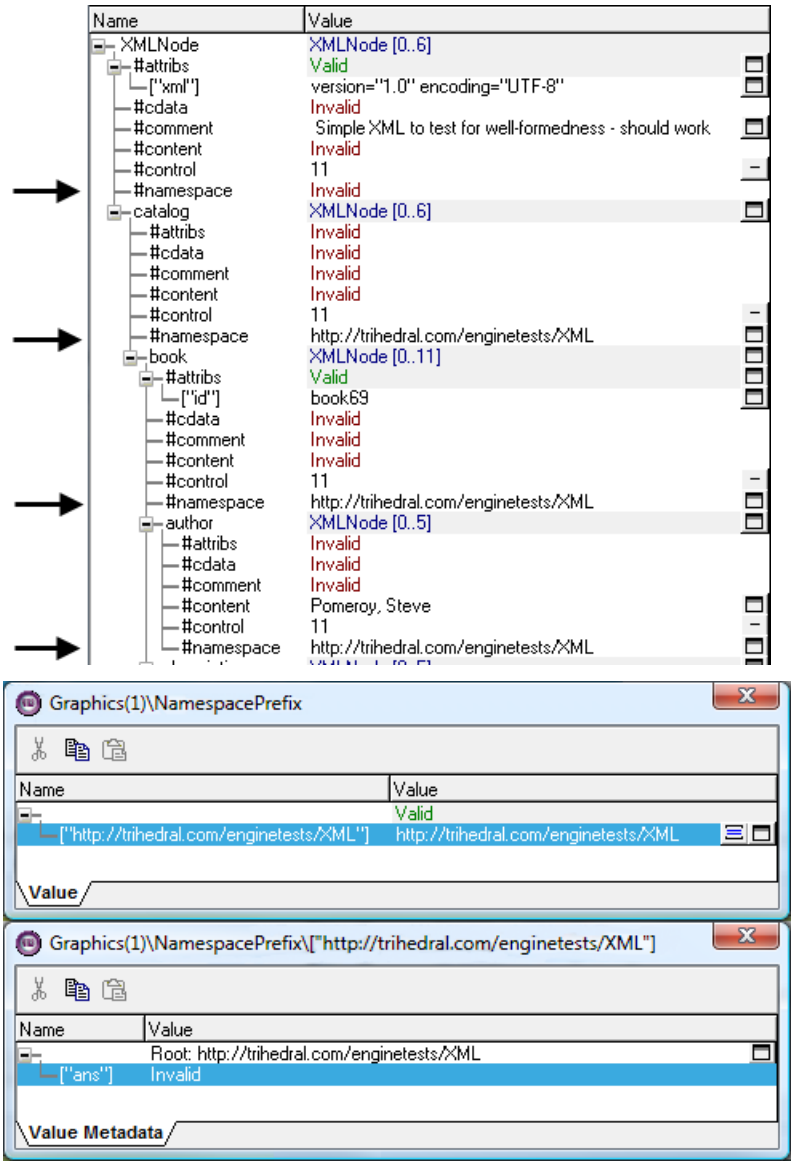
... XMLCloneNode

... XMLDeleteMember

XML Namespaces

Namespaces are used to avoid ambiguity between XML elements of the same name. An XML Namespace is often specified as a URI. Such a URI might typically look like:
http://schemas.xmlsoap.org/wsdl/http/
Namespaces are declared with an associated prefix that is then applied to a tag. When the XMLNode representation is constructed, the prefix is removed and the namespace corresponding to the prefix is set in the #namespace member of the node. When writing out an XMLNode tree, a namespace dictionary is supplied, often the same one produced when parsing the incoming XML that indicates the prefixes to be used for each namespace.
Although multiple prefixes could be used for a single namespace, on writing out the XMLNode tree only the first prefix added to a namespace dictionary is used. The following XML uses namespaces and the subsequent images show the XMLNode representation and the namespace dictionary.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Simple XML to test for well-formedness - should work -->
<ans:catalog xmlns:ans="http://trihedral.com/enginetests/XML">
  <ans:book id="book69">
    <ans:author>Pomeroy, Steve</ans:author>
    <ans:title>RPC Manual</ans:title>
    <ans:genre>SCADA Software</ans:genre>
    <ans:price>44.95</ans:price>
    <ans:publish_date>2006-01-19</ans:publish_date>
    <ans:description>A guide to VTScada RPC.</ans:description>
  </ans:book>
</ans:catalog>
```


Name
Value
XMLNode
XMLNode [0..6]
#attribs
Valid
["xml"]
version="1.0" encoding="UTF-8"
#cdata
Invalid
#comment
Simple XML to test for well-formedness - should work
#content
Invalid
#control
11
#namespace
Invalid
catalog
XMLNode [0..6]
#attribs
Invalid
#cdata
Invalid
#comment
Invalid
#content
Invalid
#control
11
#namespace
http://trihedral.com/enginetests/XML
book
XMLNode [0..11]
#attribs
Valid
["id"]
book69
#cdata
Invalid
#comment
Invalid
#content
Invalid
#control
11
#namespace
http://trihedral.com/enginetests/XML
author
XMLNode [0..5]
#attribs
Invalid
#cdata
Invalid
#comment
Invalid
#content
Pomeroy, Steve
#control
11
#namespace
http://trihedral.com/enginetests/XML

Graphics(1)\NamespacePrefix
Name
Value
Valid
["http://trihedral.com/enginetests/XML"]
http://trihedral.com/enginetests/XML
Value
Graphics(1)\NamespacePrefix\["http://trihedral.com/enginetests/XML"]
Name
Value
Root: http://trihedral.com/enginetests/XML
["ans"]
Invalid
Value Metadata

The VTScada Wizard Engine

A Wizard wraps up a configuration task into a series of steps where input validation can be made at each stage, and the user is free to go back and alter previously entered information until the Wizard is finalized. A simple, uncluttered user-interface makes the Wizard particularly suitable for guiding inexperienced users through complex procedures.
VTScada provides a Wizard Engine that manages the task of steering an implementer-supplied module through its various states. The Wizard Engine also provides some useful support functions.

Note: VTScada includes a wizard template that has been provided to assist programmers in creating their own wizards. This template is installed with the VTScada software, and can be found in the Example folder within the VTScada installation directory:
C:\VTScada\Examples\Wizard.SRC.
You can add import this module as a page in the Idea Studio (File menu >> Import). Make sure that you add the page to the menu, then close the Idea Studio and open the wizard page. The sample wizard will open in a pop-up, demonstrating several features that you can use in your own wizards. (Wizard pages are generally not added to the menu, but rather are opened explicitly, when required.)

The WizardEngine exports a constant Version property, so that scripts can handle future changes to the WizardEngine functionality. This property was introduced at version 2. The absence of this property indicates version 1.

Related Information:

...Getting Started

...Flow Direction

...Wizard Configuration Settings

...Cautionary Notes for Wizards

Related Functions:

Getting Started

A Wizard is a module that runs in its own window. For reasons that should become clear, the recommended (but not the only) way of structuring the Wizard module is to have an outer module with two states, with all the necessary initialization done on the transition between these, and a child module containing the Wizard code, which is launched as the final step. For these notes, the use of this model is assumed. The term "Parent" refers to the outer module, and "Wizard" refers to the Wizard code module.

Parent Module, parameters and variables:
If you intend to pass parameters to the Wizard, do so as you would for any module:

```
{  Receive any required parameters from the calling environment
...  }
(
  Param1;
  Param2; { etc as required. }
)
```

Several variable declarations are common to most Wizards:

```
[
  Title  = "Wizard Template"  {  Change this as required
};
  Wizard = Module             { Wizard serialization engine (a
submodule)   };
  Constant Forward    =  1;
  Constant Backward   = -1;
  Constant SPACE      =  8;
```

```
  Move                    { State change trigger
};
  NextState               { Next step in the wizard
serialization          };
  WTitle                  { Text for header bar
};
  Cancel           = 0    { Cancel flag - set when engine wishes to
cancel };
  Constant CRLF    = Concat(MakeBuff(1, 13), MakeBuff(1, 10));
```

```
  Constant LHS     = 8    { LHS of the message panel
};
  Constant TOP     = 60   { Top of the message
panel                       };
  Protected BOT            { Y coord of horiz line above
buttons            };
  Protected RHS            { RHS of the drawing
panel                       };
  Protected MID            { Centre of drawing
panel                        };
  Protected Split          { Horizontal division of drawing
panel           };
  Root                     { Object value of Wizard - useful for tag
config };
```

Window Parameters

If running the Wizard in its own window, the recommended window parameters are:

```
  Width  = 520;
  Height = 360;
  Style  = 0b1010010100000111;
```

Note that the height of the top and bottom bars of the window are fixed at 30 and 40 pixels respectively.

Alternatively, you can run a Display Manager page. To do this, the following attributes should be defined in the page's source file:

```
Constant PageWidth  = 520;
Constant PageHeight = 360;
Constant WinFlag    = 1;
Constant PageStyle  = 0;
Constant PagWinOpt  = 0b1010010100000111;
Constant NoStretch  = 1;
Constant Bitmap;
Constant Color      = -17 { Background color for page - #SYSCOLOR_
BUTTONFACE };
SecBit                    { Set page security as
required                    };
WindowCloseFlag     = 1   { Flag tells DispMgr not to close our win-
dow       };
```

Tag Configuration

In general, if the Wizard is being used to perform Tag configuration then the simplest way to do this is to start the Wizard with a Parms array parameter of the correct size for the Tag type and to fill in the Parms array as you proceed. It is usually possible to call a Tag's ConfigFolder since (with Root pointing to Self and the Parms array provided) the calling

environment is correct. If it is required to configure multiple Tags, then create multiple parameter arrays and swap them in and out of "Parms" as needed.

```
{== Sample variables for tag configuration
==================================}
{ Parameter indices - these are standard, add others as
required              }
  Constant #Name        = 0;
  Constant #Area        = 1;
  Constant #Description = 2;
  #IODevice;
  Protected TmpCfg;
  Parms;
{============== End parent module variable declarations
======================}
]
```

Parent Module Initialization

The Parent performs the initialization of resources required by the Wizard. It is also likely that in its main state, it has to monitor a Cancel flag and to tear down the structure if the flag is set.

If running as a Display Manager page, the Init state should launch the Wizard child module before running the main state, which will watch for a Cancel from the user. If running as a Window, the main state should call the Window function, passing the Wizard child module as a parameter.

The example wizard (C:\VTScada\Examples\Wizard.SRC) shows how to write code that can be used either way. The following assumes the Wizard will run in a Display Manager page.

```
WizardInit [
  If 1 WizardMain;
  [
    { Setup instance }
    SetInstanceName(Self, \SecurityManager\GetAccountID());
    { Setup some basic metrics }
    RHS = PageWidth - 8;
    BOT = PageHeight - 40;
    MID = Int(PageWidth / 2);
    Split = MID + 40;
```

If you are going to be configuring Tags, then you will likely need a parms array and various offsets into that array. You can establish these dynamically as follows...

```
    { Get a module/object handle to the required tag
type, where "SampleTag"
      is to be replaced by the module name of the type you require.
}
    TmpCfg = Scope(\Code, "SampleTag");

    { Get its parameter indices }
    #IODevice = GetDefaultValue(FindVariable("#IODevice", TmpCfg, 0,
0));
    { ... etc. }
    Parms     = New(FormalParms(TmpCFG));
```

Or, alternatively:

```
    #IODevice = GetDefaultValue(FindVariable("#IODevice", Variable
("SampleTag"), 0, 0));
    { ... etc. }
    Parms     = New(FormalParms(Variable("SampleTag")));
```

Finally, launch the wizard:

```
    { Running as a DisplayManager page - launch a Wizard }
    Wizard(Parms);
  ]
]    {===== End WizardInit ======}
```

Parent Module Main State

```
WizardMain [
  { Running as a DisplayManager page }
  If Cancel;
  [
    \DisplayManager\StopPage(Self);
  ]
]
```

Note: VTScada includes a wizard template that has been provided to assist programmers in creating their own wizards. This template is installed with the VTScada software, and can be found in the Example folder within the VTScada installation directory: C:\VTScada\Examples\Wizard.SRC.
You can add import this module as a page in the Idea Studio (File menu >> Import). Make sure that you add the page to the menu, then close the Idea Studio and open the wizard page. The sample wizard will open in a pop-up, demonstrating several features that you can use in your

---

own wizards. (Wizard pages are generally not added to the menu, but rather are opened explicitly, when required.)

Next Steps:

...Basic Wizard Engine Module

...Flow Direction

...Wizard Configuration Settings

...Cautionary Notes for Wizards

Basic Wizard Engine Module

Note: Use only the Wizard Engine methods. Do NOT add your own state transition logic for the states controlled by the Wizard Engine.

The Wizard has a few special requirements:

- The first step of the Wizard user interface must correspond to the first state in the module.
- The Wizard must launch a copy of \WizardEngine into its own scope. It will need to retain the Object value in order to access the helper modules.
- Several variables will be required to be passed to the instance of \WizardEngine for control of the actions of the Wizard.
- The Wizard must have a state named, "Finish". This is not necessarily the final state in the Wizard, but is the final step of the user interface. As many additional states as are required to perform the Wizard's completion tasks may come after the "Finish" state.
- If the Wizard is to perform Tag configuration, it may be useful for the Wizard to be instantiated with a suitable Parms array and a Root variable so that it can impersonate a tag instance.

In practice, the Wizard module starts at its first state and is then navigated between its various states by the controlling WizardEngine, which responds to the user's use of the "Next" and "Back" buttons, as well as providing the programmatic flow control. Once the Wizard reaches its

"Finish" state, and the user presses the "Finish" button, the Wizard then does whatever is required to perform its required operations. There is no going back from this point; the Wizard must complete its task. All necessary validation must be performed prior to the Wizard reaching the Finish state.
The Wizard has to launch a copy of the WizardEngine on entry to its first state; typically:

```
<
{================================ Wizard
====================================}
{ The Wizard serialization module. }
Wizard
(
  Parms;
)
[
  Protected Engine    { Instance of the WizardEngine        };
  Protected Trig      { Trigger for input field completion  };
  Protected Msg       { Current message to display          };
  {                                  }
  {  Other variables as required... }
  {                                  }
]
```

The Wizard Engine has the following parameters:

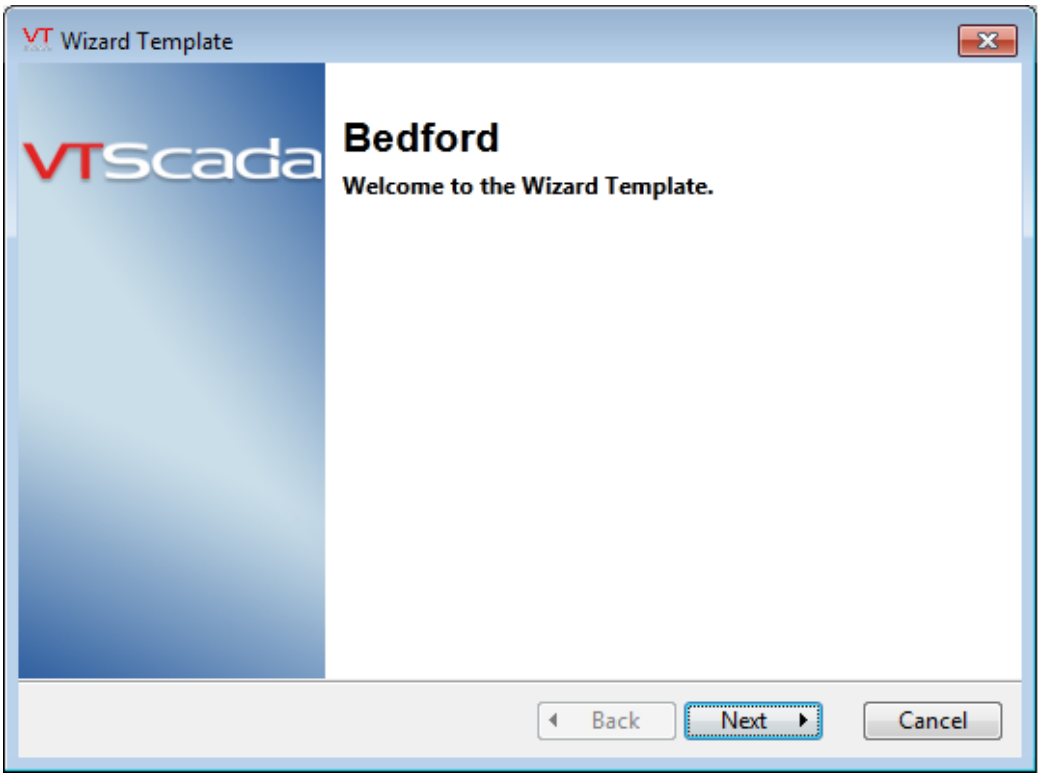
- WizardName: A text value setting the name for this particular Wizard.
- WizBmp: An image value of a graphic that will be displayed at the first and last steps of the Wizard (see the notes later in this topic for sizing information).
- pState: A pointer to a variable that is set by the WizardEngine to the name of the next state that the Wizard should execute.
- pMove: A pointer to a variable set by the WizardEngine as a trigger when the Wizard is required to change states. The actual value of the trigger provides further information, if required, about the direction of movement, and so forth.
- pClose: A pointer to caller's close flag – set by engine.
- pTitle: A pointer to the caller's current title string. This will be displayed in the header.
- LogoBmp: An image value of a graphic that will be displayed at the RHS of the header bar. If Invalid, LogoBmp defaults to \DispMgrBitmap.

- AppName: A text value that will be displayed as the initial title and at the bottom left of other screens. Defaults to the current application name if invalid.

```
Init [
  If !Valid(Engine);
  [
    Root = Self();
    Engine = Launch(Scope(\Code, "WizardEngine"), Self, Self,
                GetDefaultValue(FindVariable("Title", Self, 0, 1)),
                MakeBitmap(FileFind("C:\VTScada\Re-
sources\WizVTS.jpg," 0)),
                &nextState,
                &Move,
                &Close,
                &WTitle);
  ]
  WTitle = ConCat("This wizard is a template for a new Wizard.",
                  CRLF, CRLF, CRLF, CRLF,
                  "Press NEXT to continue");
```

```
  If Move;
  [
    Move = 0;
    ForceState(NextState);
  ]
]
```

The sample code above, running in a page titled "Wizard Template" in an application named, "Bedford," produces the following result.



Note: The example above is only a portion of a wizard module. VTScada includes a wizard template that has been provided to assist programmers in creating their own wizards. This template is installed with the VTScada software, and can be found in the Template directory within the VTScada installation directory: C:\VTScada\Example\Wizard.src.

With regards to the image above:

- The image is scaled so that its height fits the vertical space between the title bar and the bottom bar. The aspect ratio then determines the positioning of the RH panel. To avoid distortion, it is best to size the image exactly. The bottom bar is 40 pixels high.
- The large title "Bedford" comes from the "AppName" parameter.
- In this instance, the "WizardName" parameter is "Wizard Template" (obtained from the DisplayManager page name), and this string is used in the window caption and the "Welcome to..." message.
- The remainder of the text comes from the variable addressed by the pTitle parameter. Note the use of CRLF (defined as:

```
Constant CRLF = Concat(MakeBuff(1, 13), MakeBuff(1, 10));
```

to achieve vertical spacing).

Let's review the state code:

```
If Move;
[
   Move = 0;
   ForceState(NextState);
]
```

This is the basic Wizard control. When the Next button is clicked, the variable "Move" (the "pMove" parameter to the WizardEngine) will be set to non-zero, and the variable "NextState" (the "pState" parameter to the WizardEngine) will be set to the name of the next state to be used. So, the remaining code for a simple Wizard could be:

```
StateTwo [
   WTitle = "We are at the second step.";
   If Move;
   [
      Move = 0;
      ForceState(NextState);
   ]
```

```
]
ThirdStep [
   WTitle = "We are now at the third step.";
   If Move;
   [
      Move = 0;
      ForceState(NextState);
   ]
]
Finish [
   If Move;
   [
      Move = 0;
      ForceState(NextState);
   ]
]
DoLotsOfWork [
  If 1 MoreWork;
  [
.. .. .. .. ..
```

Note that a title is not required for the "Finish" step, as the WizardEngine generates standard text.

Related Functions:

Error Messages [Error]	Dead Ends [NoNext]
Skipping [SkipIf]	Dead Ends [NoBack]
Branching [Switch]	Initial Action [InitCheckBox]
Triggered Branch [ForceMove]	Final Action [FinalCheckBox]
Unconditional Branch [NextIs]	Final Processing Stage [EndControl]

Wizard API

The following methods are supported by the Wizard Engine. An example of each is provided in following topics.
Given a Wizard object, creating with a statement similar to the following:

```
    Engine = Launch(Scope(\Code, "WizardEngine"), Self, Self,
                GetDefaultValue(FindVariable("Title", Self, 0, 1)),
                MakeBitmap(FileFind("C:\VTScada\Re-
sources\WizVTS.jpg," 0)),
                &nextState,
                &Move,
                &Close,
                &WTitle);
```

Then, each of these methods would be invoked as follows:

```
Engine\Method(Parameters);
```

Error dialog

```
Error(
       Msg1 { First line of error message},
       Msg2{ Second line of error message},
       Msg3{ Third line of error message}
      );
```

Usage: Script

Causes an error message dialog to be displayed and cancels the current state change. Up to three lines of error message are displayed depending on the number of parameters. This is a subroutine message, pausing execution until the error message is acknowledged.

Final Check Box

```
FinalCheckBox(
               Value  { Initial, and returned, value for checkbox  },
               Msg    { Text message for check box                 },
               Enable { TRUE to enable the display of the checkbox }
             );
```

Usage: Steady State

Enables caller to define a checkbox for the final wizard screen, stating, "Do this when the wizard finishes".

May be used only in the at the end of the wizard process.

Force Move

```
ForceMove(
           MoveNow  { If valid TRUE, then trigger the state change
now },
           NewState { New target state name }
          );
```

Usage: Steady State

Changes the next forward state. May be used to branch to a later state if intermediate states are optional.

Initial Check Box

```
InitCheckBox(
              Value  { Initial, and returned, value for check box  },
              Msg    { Text message for check box                  },
              Enable { TRUE to enable the display of the check box }
            );
```

Usage: Steady State

Enables the caller to define a check box for the first wizard screen, saying: "Do this when the wizard starts" ...
Valid only when used at the start of the wizard.

Next Is

```
NextIs(
        NewState { New target state name }
       );
```

Usage: Steady State
Specify (change) the next state in the wizard flow. (The default is the next state in the source code file.)

No Back

NoBack(
        Inhibit { While valid & TRUE, the "Previous" button is disabled. }
       );

Usage: Steady State
May be used to prevent a return to the previous state.

No Next

```
NoNext(
        Inhibit { While valid & TRUE, the "Next" button is disabled.
}
       );
```

Usage: Steady State
May be used to prevent forward progress until the operator provides information in the current state.

Skip If

```
SkipIf(
        Condition { If TRUE, then switch to NewTarget state },
        NewTarget { Optional target for the switch.
                    Defaults to the next state. }
      );
```

Usage: Steady State
Evaluates its first parameter and, if true, causes an immediate state change to the second parameter (or the next state). Note that the current state is not added to the history (does not become the previous state).

Switch

```
Switch(
        Labels       { Array of short label for radio buttons   },
        Bmps         { Array of bitmap values for icons (32*32) },
        Descriptions { Array of descriptive texts
                       (max is 4 lines each)                    },
        Destinations { Array of target state names              },
        DefaultVal   { Optional, default initial choice         },
        BottomMargin { Optional, padding to add to bottom       }
       );
```

Usage: Steady State

Displays a set of radio buttons and sets the next state change according to the chosen option. Explanatory text and an optional icon are displayed for each button.

Trim

```
Trim(
      TextString
    );
```

Usage: Script

A subroutine that removes leading & trailing whitespace from string, returning the result.

End Control

```
EndControl(
            HideOverlay { Boolean. Set TRUE to hide the overlay
                          window while processing is underway (and
                          display a progress bar or other indicator)
                          Set FALSE when done to restore the
                          overlay window. },
            FinalTitle { The final title },
            FinalText  { Text to display at finish }
          );
```

Usage: Steady State

Controls processing options at end of Wizard flow by enabling the extended finish logic.

Setting HideOverlay, will hide the overlay window, allowing the wizard to draw (progress info etc) in the main window. All buttons are disabled

Clearing HideOverlay, will reinstate the overlay window (with optional text messages) and enable the Finish button again (Back and Cancel disabled)

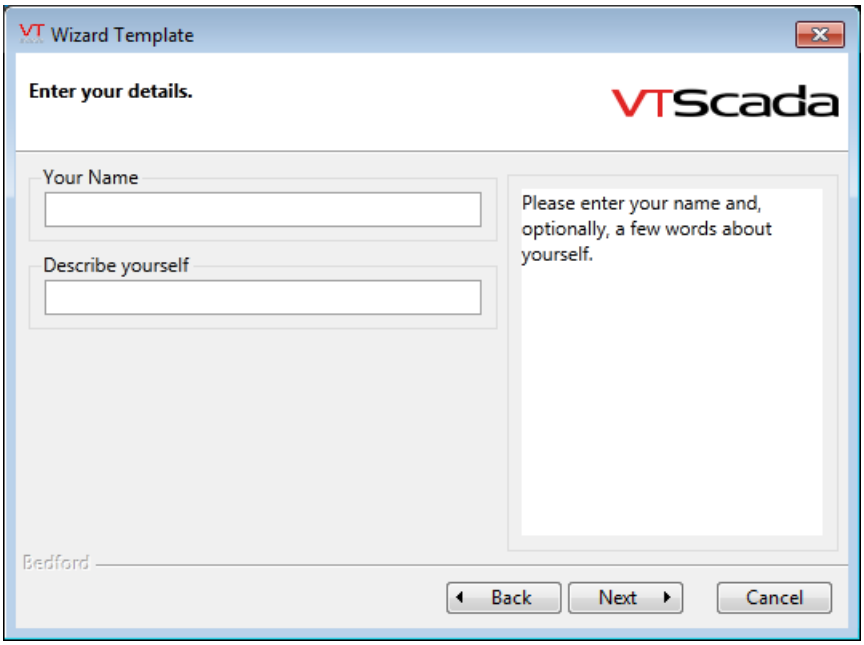
Flow Direction

The script code in the state changes is where any work resulting from a particular state needs to be done, including validation of any entered values. Additionally, when you consider Wizard actions, it becomes clear that you need to consider the direction of flow. For example, if you move forward to a particular step, and then move back to the previous step without entering anything, you do not want the Wizard to complain, whereas you do want it to complain if you try to move forward without entering anything. Therefore, the value of Move indicates the direction of travel. Forward = 1, Backward = -1, and you can write:

```
If Move;
[
   IfThen(Move == Forward,
{ Perform Validation }
```

Text Input and Output

The WizardEngine utilizes the \System\TextBox method, with which you can output multi-line text for instructions.
To collect information from the operator, you can use any of the standard input widgets, particularly members of the PTools library. With suitable code, you can also display a tab from a tag's ConfigFolder.
Remember to use and check the input trigger for any fields before changing states. Since the trigger values for the standard input widgets are all set in script, you only need one trigger variable, and you can test it for simply being valid.
A good guideline to follow for laying out Wizards is to set the input controls extending 40 pixels to the right of the mid-point, and placing helper text in a TextBox that starts 48 pixels to the right of the mid-point.
The code for this example follows the image. This state also illustrates input validation, where the user must provide an answer for the field, Your Name before they may press Next.



The code for this state follows:

```
AskDetails [
  WTitle = "Enter your details.";
  Msg = "Please enter your name and, optionally, a few words about
yourself.";
  \System\TextBox(Split + 8, BOT - 10, RHS, TOP + 17, Msg, \_Dia-
logFont, 5);
  GUITransform(0, 1, 1, 0,
               1 - LHS, TOP + 55,
               Split, 1 - (TOP + 10),
               1 { Scale whole },
               0, 0, 1, 0 { No movement; visible; reserved },
               0, 0, 0 { Not selectable },
               \DialogLibrary\PEditField(#Area, "Your Name", 4 { Text
},
                                         1 { ID }, Trig { trigger },
                                         1 { view }, 1 { Bevel }, 0 {
VAlign },
                                         1 { AlignTitle }, Invalid
{Min},
                                         Invalid {Max}, 1
{PrivNotReqd}));
  GUITransform(0, 1, 1, 0,
               1 - LHS, TOP + 110,
               Split, 1 - (TOP + 65),
               1 { Scale whole },
               0, 0, 1, 0 { No movement; visible; reserved },
               0, 0, 0 { Not selectable },
               \DialogLibrary\PEditField(#Description, "Describe your-
self",
                                         4 { Text }, 2 { ID }, Trig {
trigger },
                                         1 { view }, 1 { Bevel }, 0 {
```

```
VAlign },
                                          1 { AlignTitle }, Invalid
{Min},
                                          Invalid {Max}, 1
{PrivNotReqd}));

  If Move && PickValid(Trig != 0, 1);
  [
    IfThen(Move == Forward,
      Engine\Trim(&Parms[#Area]);
      Emsg = Invalid;
      IfThen(PickValid(Parms[#Area] == "", 1),
        Emsg = "I do need to know your name.";
      );
      Engine\Error(Emsg);
    );
    Move = 0;
    ForceState(NextState);
  ]
]
```

Cleaning Up Input [Trim]

The Wizard Engine provides a subroutine method, "Trim," which will remove leading and trailing spaces from a string. Trim takes a single parameter, a pointer to a string, which is updated in place by the Trim method.

```
  If Move;
  [
    IfThen(Move == Forward && PickValid(Trig != 0, 1),
      Engine\Trim(&Username); {trim leading & trailing whitespace}
      IfElse(PickValid(Username, "") == "",
        Engine\Error("A username is required");
        … … …
```

Error Messages [Error]

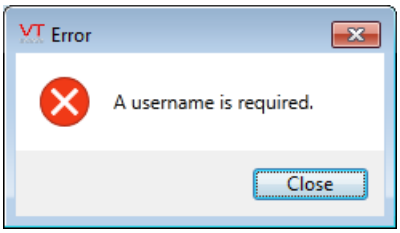
If validation is performed when leaving a state, then there is a need to be able to issue an error message to the user and cancel the move to the next state. This function is provided by the WizardEngine's "Error" method.

The "Error" method takes one, two, or three parameters, being the first, second, and third lines of a 4BtnDialog. Calling "Error" will cause the dialog to be displayed and the state move cancelled.

> Note: You must call "Error" before you call "ForceState".

Example:

```
If Move;
[
IfThen(Move == Forward && PickValid(Trig != 0, 1),
  IfElse(PickValid(Username, "") == "",
    Engine\Error("A username is required.");
  { Else }
    IfElse(PickValid(Password, "") == "",
      Engine\Error("A password is required.");
    { Else }
      IfThen(!Valid(PasswordConfirm) || (Password !=
PasswordConfirm),
        Engine\Error("Passwords do not match.");
      );
  ));
);
Move = 0;
ForceState(NextState);
]
```



Skipping [SkipIf]

Sometimes it is necessary to skip a particular step, if some condition is true. The WizardEngine method "SkipIf" performs this function. If its first parameter evaluates to true, then "Move" and "NextState" are immediately set to cause transition to the next state in the current direction of travel. Since it may also be necessary to skip validation rules, the "Move" variable indicates that skipping is taking place – it has the value "2" if skipping forward and -2 if skipping backward.
The second (optional) parameter enables the destination state to be specified. This method is either a subroutine or a steady-state call.

```
GetTagName [
   Engine\SkipIf(Valid(TagName), "GotTagName");
   GUITransform(0, 1, 1, 0,
… … …
```

Branching [Switch]

More useful than simple skipping is to present the user with a set of choices, and to branch the flow as requested. The WizardEngine provides

a method named, "Switch" for this purpose. The parameters to "Switch" are:

- Labels: An array of labels for the radio buttons.
- Bmps: An array of image values for the icons (32*32 images).
- Descriptions: An array of descriptions (up to four lines of text can be accommodated).
- Destinations: An array of state names.
- DefaultVal: An optional parameter that is the value (0..2) of the initial selection.

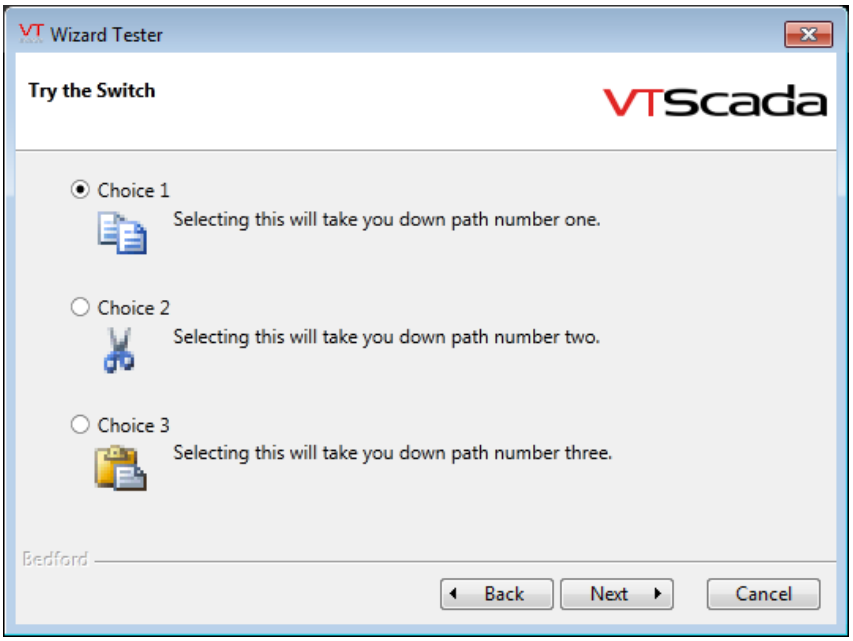
A maximum of three choices can be accommodated. Assuming that suitable variables were declared and initialized:

```
Tasks         = New(3);
TaskDescs     = New(3);
TaskBmps      = New(3);
TaskSwitch    = New(3);
Tasks[0]      = "Choice 1";
Tasks[1]      = "Choice 2";
Tasks[2]      = "Choice 3";
TaskBmps[0]   = MakeBitmap(FileFind("C:\VTScada\Resources\Copy.png,"
0))
TaskBmps[1]   = MakeBitmap(FileFind("C:\VTScada\Resources\Cut.png,"
0))
TaskBmps[2]   = MakeBitmap(FileFind("C:\VTScada\Resources\Paste.png,"
0))
TaskDescs[0]  = "Selecting this will take you down path number one.";
TaskDescs[1]  = "Selecting this will take you down path number two.";
TaskDescs[2]  = "Selecting this will take you down path number
three.";
TaskSwitch[0] = "Switch1";
TaskSwitch[1] = "Switch2";
TaskSwitch[2] = "Switch3";
```

Then the state for the branch might appear as follows:

```
SelectTask [
  WTitle = "Select Required Task";
  Task = Engine\Switch(Tasks, TaskBmps, TaskDescs, TaskSwitch,
SvTask);
  If Move;
  [
    SvTask = PickValid(Task, 0);
    Move = 0;
    ForceState(NextState);
  ]
]
```

Example:



"Switch" returns the value (0..2) selected.

Triggered Branch [ForceMove]

Sometimes it may be necessary to force a change of Wizard state without waiting for the user to click the "Next" button (for example, the user may double-click some item in a list box and expect some response). The method "ForceMove," which takes two parameters, performs this task. The second parameter enables the destination state to be specified. The first parameter triggers the state switch. The method is a steady-state call and only works in a forwards direction.

```
PersonDetails [
   Engine\ForceMove(DClick, "EditPerson");
   If Move;
… … …
```

Unconditional Branch [NextIs]

When flow has been branched using the "Switch" method, it is likely that there will be a need to use an unconditional branch to re-merge the flow. The WizardEngine provides a method named, "NextIs," which changes the destination state for the step from the default (next state), to the state whose name is provided as the first parameter of the "NextIs" method. The method is a steady-state call.

Example:

```
Engine\NextIs("StateName");
```

Dead Ends [NoNext]

There are situations when validation determines that the user can go no further down his chosen path until he goes back and changes something. The WizardEngine provides a method named, "NoNext," which causes the "Next" button to be disabled. This method takes a single parameter which should be set to TRUE to disable the "Next" button. The method is a steady-state call.

Dead Ends [NoBack]

If used, the method NoBack will inhibit the Back button until the next (forward) state change. Generally, this is not a desirable action in a wizard. Before using this feature, please consider the options provided by the EndControl method first.

Initial Action [InitCheckBox]

There is sometimes a need to give the user some option about some action when the Wizard starts. The WizardEngine provides a method InitCheckBox for this purpose. It takes three parameters:

- Value the initial value for the check box, and the returned value
- Message the message/label attached to the check box
- Enable if TRUE, then the check box is displayed.

This method should only be called from the initial state (it will be disabled otherwise).

```
Init [
  { Initial step }
  Engine\Initcheck box(tellme, "Tell me about alternatives to this
Wizard", Valid(tellme));
  If Move;
  [
    ForceState(NextState);
    IfThen(Move == Forward,
.. .. ..
```

Final Action [FinalCheckBox]

There is sometimes a need to give the user options for actions to perform when the Wizard completes. The WizardEngine provides a method named, "FinalCheckBox" for this purpose. It takes the following three parameters:

- Value: The initial value for the check box, and the returned value.
- Message: The message/label attached to the check box.
- Enable: If true, then the check box is displayed.

This method should only be called from the "Finish" state; it will otherwise be disabled.

```
Finish [
{ Final Step }
   Engine\Finalcheck box(RunNow, "Run the report when the wizard finishes", Valid(RunNow));
   If Move;
   [
      ForceState(NextState);
      IfThen(Move == Forward,
```

Final Processing Stage [EndControl]

The default flow of the wizard is that the user can move freely back and forth between the Init state and the Finish state. When the user clicks the Finish button, there is no more interaction with the user and the wizard window closes once any residual processing is complete.
If the final processing is likely to take some time, it is good user interface design to present a progress indication while processing completes, and then have a final landing point when everything is finished.
The method EndControl will alter this final stage. It takes three parameters:

- HideOverlay valid and non-zero to hide the final window overlay
- FinalTitle a text string to override the default title on the final screen
- FinalText a text string to override the default text on the final screen

Notes:

- If any of these parameters is invalid, that parameter is ignored
- The two character sequence "^W" in the FinalTitle will be replaced by the Wizard's name

This method is as follows:

When moving forward from the Finish state (i.e. the Finish button has been clicked) call EndControl(1) to hide the overlay window. This will return the wizard display to be that which normally shows between the start and finish states. Your wizard code can now draw text, progress bars etc to keep the user informed.

You can change states (using the ForceState method to do this) and this window will remain on display.

When all processing is complete, change to a final state and call EndControl(0, "Title", "Message"). The window will change to the usual final screen, except that you can (optionally) replace the title and message text. The only button that is enabled is the Finish button and when this is clicked, you should terminate the wizard.

The following code sample is an example of using these features. In this example, the normal finish screen shows the Finalcheck box. If it is checked, then the wizard enters the two stage finish.

```
Finish [
  Msg = "This is the final state in the Wizard." + CRLF + CRLF +
        "However, if you wish to test the extended finish " +
        "features, then tick the check box.";
  Engine\EndControl(Invalid, Invalid, Msg);
  Engine\FinalCheckBox(ExtendedFinish, "Use extended finish", 1);
  Engine\NextIs(ExtendedFinish ? "LongFinish" : "AllDone");
  If Move;
  [
    Move = 0;
    ForceState(NextState);
  ]
]

LongFinish [
  WTitle = "Extended Finish";
  Msg = "We are waiting here for 10 seconds to simulate ongoing back-
ground tasks." + CRLF + CRLF +
        "At the end of that time we will automatically advance to the
end of the Wizard.";
  \System\TextBox(LHS, TOP + 65, RHS, TOP + 17, Msg, \_DialogFont,
5);
  Engine\EndControl(1 {Hide Overlay});
  Engine\ForceMove(TimeOut(1, 10), "DoneDone");
```

```
  If Move;
  [
    Move = 0;
    ForceState(NextState);
  ]
]

DoneDone [
  Engine\EndControl(0 {Unhide overlay}, "The ^W is totally fin-
ished!", "We really are all done now." + CRLF + "Thank you for your
patience.");
  If Move;
  [
    Move = 0;
    ForceState(NextState);
  ]
]

AllDone [
  If !Cancel;
  [
    { All done now, let's get out of here }
    Cancel = 1;
  ]
]
```

Wizard Configuration Settings

There are 5 text labels used by the Wizard Engine. These can be added to your application's Settings.Startup file.

- WizardWelcomeTitle = Welcome to the ^W.
- WizardFinishText1 = The Wizard has acquired all necessary information.
- WizardFinishText2 = Press FINISH to complete the operation, BACK to change parameters or
- WizardFinishText3 = CANCEL to abort the operation without making any changes.
- WizardFinishTitle = The ^W is ready.

Note that the sequence "^W" in WizardWelcomeTitle and WizardFinishTitle will be replaced by the name of the Wizard.

Cautionary Notes for Wizards

There are several cautionary notes that you should keep in mind when creating a Wizard.

Loops

It is possible to use the "Switch," "ForceMove," and "SkipIf" methods to create a Wizard structure where the user can go round a loop several times. The WizardEngine attempts to recognize when a sequence of steps is being repeated, and to delete the duplicates from the history, so that when you use the "Back" button, you just go round the loop once.

State Changes

Do not be tempted to set your own state changes in your Wizard code. The WizardEngine methods must be called for all state changes whilst the "Next" and "Back" buttons are active; however, once you have moved to the state after your "Finish" state, then you are in control, and not only must you control all state changes, but you must also arrange termination once you are done (i.e. set the "Cancel" flag). Note that if you use the EndControl method to allow progress to be displayed after the finish state, then you must continue to use Wizard logic to change states until you call EndControl(0).

Validation

You must fully validate all user input before reaching the "Finish" state. when you reach this state and the user clicks "Finish," you must complete the programmed task. There is no going back, and no opportunity for error exits.

Note: Remember to use the Trim method before testing for null input .

General Reference

Technical notes for:

Colors, Fonts and Graphics

...VTScada Color Palette

...Constants for System Colors

...Color Theme Definition

...Fill Patterns

... Font Character Sets

...GUI Object Return Codes

...Line Types

Time and Date

...predefined Date Codes

...Date Formatting Strings

...predefined Time Formats

...Time Formatting Codes

...VTScada and Time Synchronization

Value Types and Language Support

... SQL Data Types

...Database Type Codes used in the ODBC Manager

...VTScada Value Types - Numeric Reference

...Value and Type Conversions

...Language Support

...Using a Non-English Character Set

Help Files, Function Support, Other...

ASCII Constants

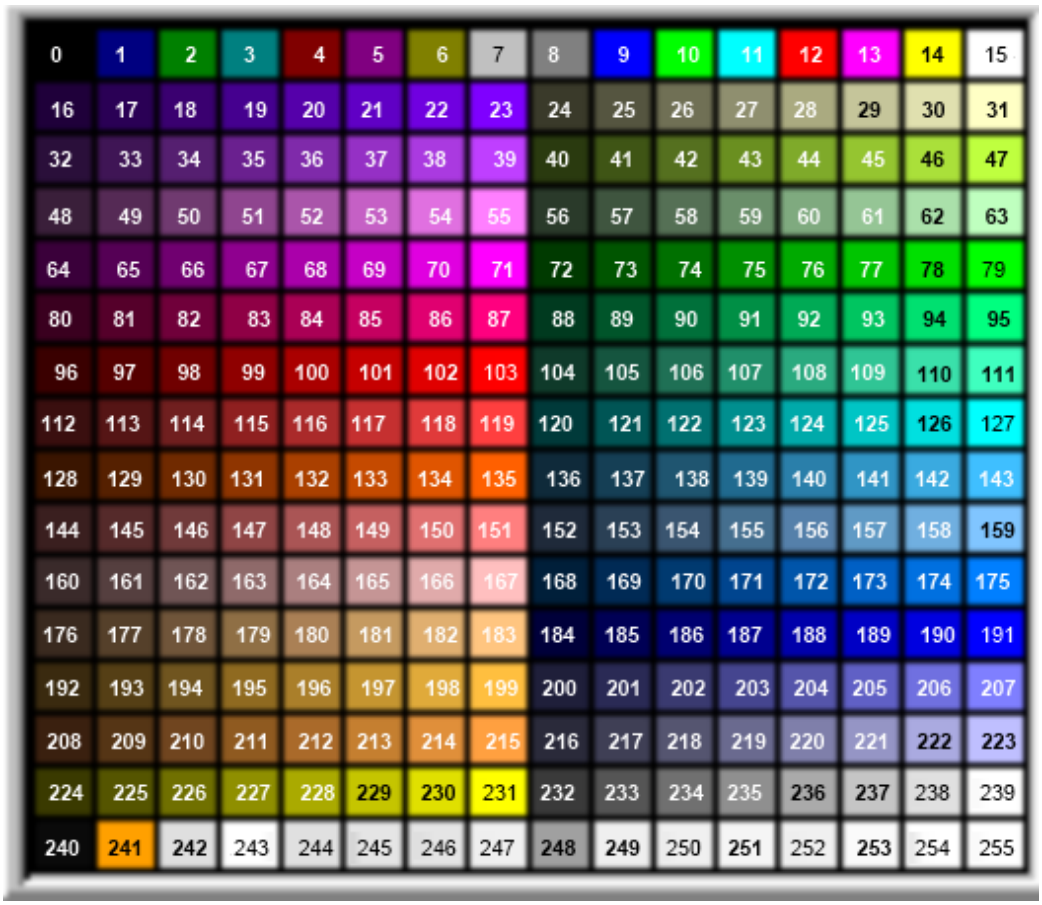
The following constants are defined at the \System layer. In code that parses user-input strings, you should use these constants rather than creating your own buffers for common key values or key combinations.

Name of Constant	Contents	Description
CR	MakeBuff(1, 13)	Carriage return
ESC	MakeBuff(1, 27)	Escape key
LF	MakeBuff(1, 10)	Line feed
NULL	MakeBuff(1, 0)	Null byte
TAB	MakeBuff(1, 9)	Tab key
CRLF	Concat(MakeBuff(1, 13), MakeBuff(1, 10))	CR/LF pair
UpArrow	Concat(MakeBuff(1, 253), MakeBuff(1, 0x48))	Up arrow key
DownArrow	Concat(MakeBuff(1, 253), MakeBuff(1, 0x50))	Down arrow key
LeftArrow	Concat(MakeBuff(1, 253), MakeBuff(1, 0x4B))	Left arrow key
RightArrow	Concat(MakeBuff(1, 253), MakeBuff(1, 0x4D))	Right arrow key
AltLeftArrow	Concat(MakeBuff(1, 253), MakeBuff(1, 0x9B))	Alt and left arrow
AltRightArrow	Concat(MakeBuff(1, 253), MakeBuff(1, 0x9D))	Alt and right arrow
SUpArrow	Concat(MakeBuff(1, 253), MakeBuff(1, 0xB8))	Shift and up arrow
SDownArrow	Concat(MakeBuff(1, 253), MakeBuff(1, 0xC0))	Shift and down arrow

PageUp	Concat(MakeBuff(1, 253), MakeBuff(1, 0x49))	Page up key
PageDown	Concat(MakeBuff(1, 253), MakeBuff(1, 0x51))	Page down key
HomeKey	Concat(MakeBuff(1, 253), MakeBuff(1, 0x47))	Home key
EndKey	Concat(MakeBuff(1, 253), MakeBuff(1, 0x4F))	End key
DeleteKey	Concat(MakeBuff(1, 253), MakeBuff(1, 0x53))	Delete key
SPageUp	Concat(MakeBuff(1, 253), MakeBuff(1, 0xB9))	Shift and page up
SPageDown	Concat(MakeBuff(1, 253), MakeBuff(1, 0xC1))	Shift and page down
SHomeKey	Concat(MakeBuff(1, 253), MakeBuff(1, 0xB7))	Shift and home
SEndKey	Concat(MakeBuff(1, 253), MakeBuff(1, 0xBF))	Shift and end
CtrlPageUp	Concat(MakeBuff(1, 253), MakeBuff(1, 0x84))	Ctrl and page up
CtrlPageDown	Concat(MakeBuff(1, 253), MakeBuff(1, 0x76))	Ctrl and page down
CtrlHome	Concat(MakeBuff(1, 253), MakeBuff(1, 0x77))	Ctrl and home
CtrlEnd	Concat(MakeBuff(1, 253), MakeBuff(1, 0x75))	Ctrl and end
CtrlBKey	MakeBuff(1, 2)	Ctrl and B
CtrlCKey	MakeBuff(1, 3)	Ctrl and C
CtrlDKey	MakeBuff(1, 4)	Ctrl and D
CtrlIKey	Concat(MakeBuff(1, 253), MakeBuff(1, 0xB5))	Ctrl and I
CtrlLKey	MakeBuff(1, 12)	Ctrl and L
CtrlNKey	MakeBuff(1, 14)	Ctrl and N
CtrlOKey	MakeBuff(1, 15)	Ctrl and O
CtrlUKey	MakeBuff(1, 21)	Ctrl and U
CtrlVKey	MakeBuff(1, 22)	Ctrl and V
CtrlXKey	MakeBuff(1, 24)	Ctrl and X
CtrlYKey	MakeBuff(1, 25)	Ctrl and Y
CtrlZKey	MakeBuff(1, 26)	Ctrl and Z

VTScada Color Palette

Colors in VTScada have been specified using RGB values since the release of version 10.2. Prior to that version, the color palette, described in this topic, provided the full range of colors available for use.
All VTScada functions that require a color value will still recognize values from the color palette. Use the following chart as a guide when selecting which number to use.



Color Theme Definition

You can add your own color themes to VTScada. These are stored in the Setup.INI file, within the [Themes] section.

Themes are defined by numeric values for Hue, Saturation, Brightness and Contrast in that order, following the theme name. The scale for those values is based on the tools that were available at the time that themes were introduced, and do not use plain HSL values. For reference, note that the Plum theme with values of 0,1,1,1 is the base color.

For example:

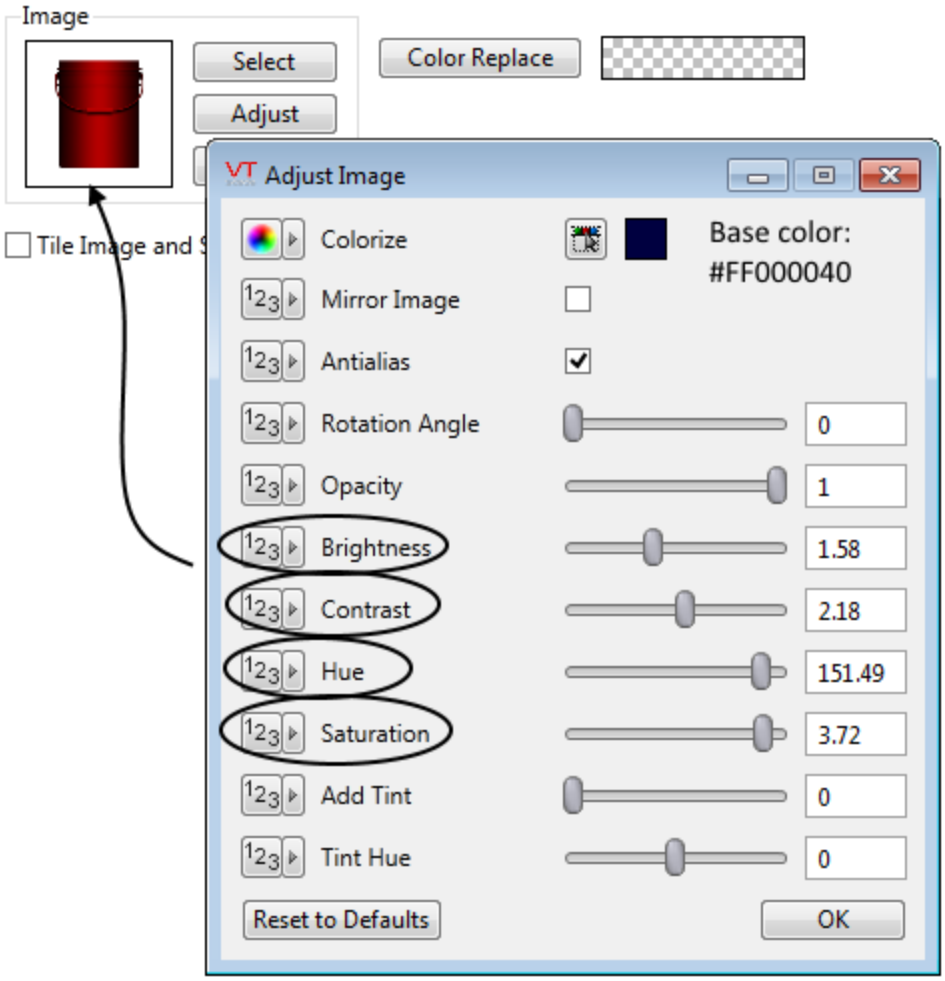
```
[Themes]
Theme = Grey,0,0,1.1,1
Theme = Navy,-15,2,0.7,1
Theme = Burgundy,110,2,0.5,1
```

Hue ranges from -180 to +180 and represents the departure from the color <FF343468> (a shade of purple).
Color saturation, brightness and contrast all range from zero to four.

The following example describes the best available method for defining a new theme.

1. Open the Idea Studio and add a grayscale image to the page.
   The choice must be an image, not a widget or shape.
2. Open the properties dialog for the image.
3. Click the Adjust button to open the Adjust Image dialog.
4. Use the Colorize control to set the base color of the image to RGB value FF343468. (Red 53, Green 52, Blue 104, Opacity 255)
5. Drag the Hue and Saturation sliders until the color of the sample image changes to approximately what you want.
6. Drag the Brightness and Contrast sliders to make fine adjustments to the sample image.
7. Note the values for Hue, Saturation, Brightness and Contrast in order.
8. Edit your Setup.INI file to add a new entry.

Remember that you will need to stop all your applications and restart VTScada itself before changes to Setup.INI take effect.



Setting the values.

The entry in Setup.INI:

```
Theme = AutumnGlow,151.5,3.72,1.58,2.18
```

The resulting theme. The high contrast setting is most clear in the controls along the bottom.

Constants for System Colors

The following constants should be used in place of their numeric values for any function that will accept a system color designation.

Constant	Value
#SYSCOLOR_SCROLLBAR	-2
#SYSCOLOR_DESKTOP	-3
#SYSCOLOR_ACTIVEBAR	-4
#SYSCOLOR_INACTIVEBAR	-5
#SYSCOLOR_MENUBACK	-6
#SYSCOLOR_WINBACK	-7
#SYSCOLOR_WINFRAME	-8

#SYSCOLOR_MENUTEXT	-9
#SYSCOLOR_WINTEXT	-10
#SYSCOLOR_ACTIVETEXT	-11
#SYSCOLOR_ACTIVEBORDER	-12
#SYSCOLOR_INACTIVEBORDER	-13
#SYSCOLOR_MDIBACK	-14
#SYSCOLOR_SELECTEDBACK	-15
#SYSCOLOR_SELECTEDTEXT	-16
#SYSCOLOR_BUTTONFACE	-17
#SYSCOLOR_BUTTONSHADOW	-18
#SYSCOLOR_GRAYEDTEXT	-19
#SYSCOLOR_BUTTONTEXT	-20
#SYSCOLOR_INACTIVETEXT	-21
#SYSCOLOR_BUTTONHLITE	-22

Integrating Custom Help Files into VTS

VTScada provides you with the power to integrate custom help files into your VTScada applications. With your own help file, you might choose to:

- Link individual tag instances to topics in a custom help file.
- Link pages to topics in a custom help file.
- Link pages, widgets or user-defined tags to any of the 100 topic files that have been set aside in the VTScada help system for customer use. You may change the text in these files as required.
- Override the default VTScada help files so that your custom help file opens when your OEM-based application is selected in the VAM and the VAM's Help button is clicked.

VTScada provides support for custom help files in a variety of formats, but not for all possible formats. The Microsoft® .HLP and .CHM formats are fully supported. HTML-format help files are produced by many help authoring tools, but each uses its own code for context sensitive help. It is not possible for VTScada to work with every format. Support is provided for both the Doc-To-Help® DotNet format and for the Flare® HTML5 help format. (Both DocToHelp and Flare are products of MadCap Software Ltd.)

Add your topics to the built-in VTScada help system.

- One hundred topics have been set aside for you to edit within the VTScada help system. Each is named for the mapping ID value assigned according to the pattern, UserTopic100, UserTopic101, etc. Instructions for finding and editing these topic files are provided in User-Topics in the VTScada Help Folder.

Create your custom help file or system and assign topic mapping ID values:

- You will need to obtain a third-party help authoring tool. For .HLP and .CHM format help files, there are many programs to choose from and prices vary considerably. If you intend to create a DotNet or HTML5 format help system, only DocToHelp® and Flare® are supported by VTScada.
- Topic mapping ID values are typically created within the help authoring tool and assigned to topics there. VTScada's numbering system begins at 10,000. In addition, values 100 through 200 are reserved for user-defined topics that you may add to the VTScada help files. To avoid any conflict with the VTScada ID values, create mapping IDs that are less than 10000, excluding those in the 100 to 200 block.

Install your custom help file:

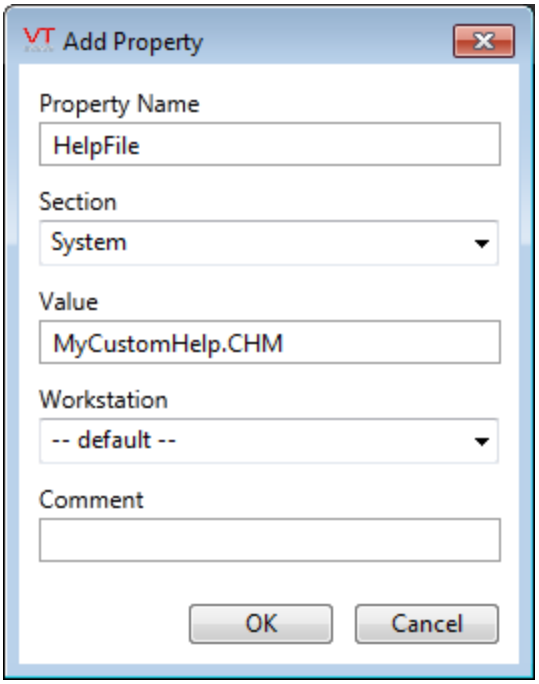
- If using the .HLP format, your help compiler will generate several files. Send both the .HLP file and the .CNT file with your application. Save both files to the VTScada installation folder, not to the application folder.

- If using the .CHM format, send only the .CHM file. Save this to the VTScada installation folder, not to the application folder.
- If using the DotNet format or the HTML5 format, copy the folder structure to a new sub-folder that you create within your VTScada installation folder (ex: C:\VTScada\MyHelpFolder)

Link tag instances to topics in your help file

Note: If using a UserTopic that you have edited in the VTScada help system, skip steps 1 and 2.

1. Use the advanced mode of the Edit Properties dialog to create a local copy of the property, HelpFile.

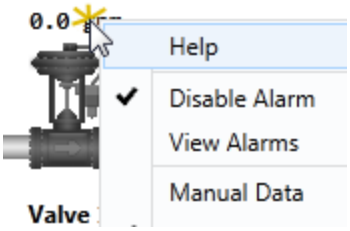


2. Set the value of that property as follows:
    - If using the .HLP or .CHM format, set the property to the name of the file, including the extension. It is assumed that the file will be stored in the VTScada installation folder, not in the application folder.
    - If using the NetHelp (DocToHelp) format, set the value of HelpFile to "MyHelpFolder\NetHelp", replacing "MyHelpFolder" with the name of the folder you created to store the help files within.

- If using the HTML5 (Flare) format, set the value of HelpFile to "MyHelp-folder\MadCapWebHelp", again replacing "MyHelpFolder" with the name of the folder you created to store the help files within.

3. Open the properties dialog of a tag instance.
4. Set the Help Key property to the value of topic mapping id withing your help file, matching a topic to the tag.

Operators can now right-click on widgets linked to that tag, then click Help in the menu that appears, in order to view your topic.



Link pages to topics in a custom help file

To create a link between your page and a topic in a custom help file, edit the page's source code, then import file changes. You can define the help file and a specific topic for any page by adding the following line within the module's Main state:

```
SetHelp(Self(), Help File Name, Mapping ID value);
```

Replace the parameter, "Mapping ID value" with the ID value in your help file that matches the topic to be opened when an operator presses F1. The help file name parameter should be replaced using the same rules as for the HelpTopic application property:

- If using the .HLP or .CHM format, set the parameter to the name of the file, including the extension. It is assumed that the file will be stored in the VTScada installation folder, not in the application folder.
- If using the NetHelp (DocToHelp) format, set the parameter to "MyHelpFolder-\NetHelp", replacing "MyHelpFolder" with the name of the folder you created to store the help files within. Include the quotation marks.
- If using the HTML5 (Flare) format, set the parameter to "MyHelp-folder\MadCapWebHelp", again replacing "MyHelpFolder" with the name of the folder you created to store the help files within. Include the quotiation marks.

- If using a UserTopic that you have edited in the VTScada help system, set the parameter to \DevHelpFile. Do not add quotation marks.

Override the default help file for the VAM's help button

A legacy feature of VTScada is the ability to have the Help button in the VAM open your help file instead of the default. Your help file will open only if the operator has first selected (but not necessarily started) an application that is based on an OEM layer rather than the VTScada library layer. Given that requirement, and the fact that it is rare for an operator to have access to the VAM, let alone be selecting applications, then clicking the help button, the following information is provided mostly for interest's sake.

1. Stop all applications and stop VTScada.
2. Open the file, Setup.INI, in the VTScada installation folder.
3. Edit the OEM section property, OEMHelp, setting its value according to the rules described twice already in this topic. Do not use quotation marks, regardless of whether you are setting the value to MyHelpFile.CHM, MyHelpFolder\NetHelp or MyHelpFolder\MadCapWebHelp.
4. Restart VTScada.
5. Select an application that was built on an OEM layer.
6. Click the VAM's help button. Your help file should open to the default welcome page.

User-Topics in the VTScada Help Folder

One hundred mapping ID values have been set aside in the VTScada help system, which you can use to link to topics you create. These ID values can be used with your pages or with individual tag instances. When an operator presses F1 while viewing your page, or right-clicks on a tag-linked widget and opens the Help option, your topic will be displayed. You must create the topic files, using the provided template as a guide. The topic files must be saved in the VTSHelp\Content\UserTopics folder. Each topic must be named according to the pattern, "UserTopic100.htm" where the numeric portion of the name (100 in this example) matches

the ID value set aside for the topic. You will use that ID for your page or tag. One hundred ID values have been set aside, therefore your topics will be named, "UserTopic100.htm" through "UserTopic199.htm".

Note: These user topics cannot be displayed in the menu of the VTScada help system, nor will your content be found in response to a search in the help system. They can be displayed only in response to an F1 request.

Edit your topic files:

1. Assuming that you have installed VTScada in the folder, C:\VTScada, navigate to the folder, C:\VTScada\VTSHelp\Content\UserTopics.
2. Copy the file, UserTopicTemplate.htm to UserTopicN.htm where N is a number from 100 to 199.
   Start with UserTopic100.htm and work through the set in order.
3. Use a text editor, or a web page editor to open the new file, UserTopicN.htm. Do not use MS Word, or any other word processing program that will add its own formatting characters.
4. Search for the keyword, "+++Start"
   The first 300 or so lines in the topic are required to load the CSS files, JavaScript, menu and other parts of the VTScada help system. Do not edit anything before "+++Start".
5. Replace the text between the <H1> header tags. Do not use headers other than <H1>.
6. The content that you are replacing includes HTML tags and CSS style tags that show styles used in the VTScada help system. You should use these as needed for your own text and images.
   All the text from "+++Start" through to "end---", should be replaced.
   "+++Start" and "end---" should be removed.
7. Save your file.

To link a tag to a user topic you created:

1. Select the topic file and note the numeric portion of the name.
2. Open the tag properties dialog and enter the number from the topic into the Help Search Key field of the ID tab.
3. Close the tag properties dialog.
4. To test, right-click on any widget linked to the tag, then click on "Help" in the menu that opens. The matching topic should open in your default browser.

To link a page to a user topic:

1. Select the topic file and note the numeric portion of the name.
2. Open the source code of the page to be linked to this topic.
3. In the Main state, add the following line of code, replacing the number with the one from the file.

```
SetHelp(Self(), \DevHelpFile, 101);
```

4. Click the Import File Changes button for this application in the VAM.

When updating your copy of VTScada, only the template file will be replaced. Your own topic files will remain but, within these the top menu of VTScada help topics may be out of date. You can bring your topics up to date by copying everything before the "+++Start" marker and after the "End---" marker from the template file, replacing the same within your UserTopicN.htm files.

Database Type Codes used in the ODBC Manager

The following numeric values are used by various functions in the library to select formatting characteristics appropriate to each database type.

Value	Meaning
0	MS SQL

1	MS Access
2	Oracle
3	MySQL
4	SyBase

predefined Date Codes

Use any of the following numeric date codes to format a data as shown. If you require a custom format, you can build one using the text codes shown in Date Formatting Strings.
Dates using these codes will always be displayed in English, regardless of system configuration.
All examples showing Monday, August 13, 2012.

Note: Use only the number in the first column - the second two describe the result of the code in the first column. They are not codes that you can use in the function.

Date Code	Example	Description
0		no date
1	120813	yyMMdd
2	08/13/12	MM/dd/yy
3	08-13-12	MM-dd-yy
4	Aug 13, 2012	MMM d, yyyy
5	August 13, 2012	MMMM d, yyyy
6	13 Aug 12	dd MMM yy
7	13 Aug 2012	dd MMM yyyy
8	13/08/12	dd/MM/yy
9	13-08-12	dd-MM-yy
10	13Aug12	ddMMMyy

11	13Aug2012	ddMMMyyyy
12	Aug 13/12	MMM d/yy
13	20120813	yyyyMMdd
14	08/13	MM/dd
15	08-13	MM-dd
16	08/12	MM/yy
17	08-12	MM-yy
18	08/2012	MM/yyyy
19	08-2012	MM-yyyy
20	Aug 13	MMM d
21	August 13	MMMM d
22	Aug 2012	MMM yyyy
23	August 2012	MMMM yyyy
24	13/08	dd/MM
25	13-08	dd-MM
26	Aug	MMM
27	August	MMMM
28	12-08-13	yy-MM-dd
29	12/08/13	yy/MM/dd
30	2012-08-13	yyyy-MM-dd
31	2012/08/13	yyyy/MM/dd
32	12-W35-01 ([1])	yy-WeekOfYear-DayOfWeek ([2])
33	12/W35/01	yy/WeekOfYear/DayOfWeek
34	2012-W35-01	yyyy-WeekOfYear-DayOfWeek
35	2012/W35/01	yyyy/WeekOfYear/DayOfWeek
36	12226	yyDayOfYear
37	2012226	yyyyDayOfYear

38	12-226	yy-DayOfYear
39	12/226	yy/DayOfYear
40	2012-226	yyyy-DayOfYear
41	2012/226	yyyy/DayOfYear
42	Mon, 13 Aug 2012	ddd, d MMM yyyy

([1]) Week Of Year is preceded by the character W.

([2]) "DayOfWeek" and "WeekOfYear" are descriptions rather than format codes that you could use.

Related Information:

predefined Time Formats - Use for displaying time.

Date - The function that uses the codes listed above.

Date Formatting Strings

To build a custom data format, assemble the following format codes into text strings. Format codes are case-sensitive. Text strings are always enclosed in double quotation marks.

Dates using these formatting strings will be displayed in the language of the user's locale.

Before building a date format with these strings, review the list of pre-defined Date Codes.

Format Code	Description
"d"	Day of month as digits with no leading zero for single-digit days.
"dd"	Day of month as digits with leading zero for single-digit days.
"ddd"	Day of week as a three-letter abbreviation.
"dddd"	Day of week as its full name.
"g"	B.C. or A.D.
"M"	Month as digits with no leading zero for single-digit months.
"MM"	Month as digits with leading zero for single-digit months.

"MMM"	Month as a three-letter abbreviation.
"MMMM"	Month as its full name.
"y"	Year as last two digits, but with no leading zero for years less than 10.
"yy"	Year as last two digits, but with leading zero for years less than 10.
"yyyy"	Year represented by full four digits.

Example:

```
Date(Today(), "dddd MMM dd, yyyy")
```

... yields, "Thursday Aug 28, 2008"

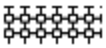
Related Information:

Time Formatting Codes - Use for displaying time.
Date - The function that uses the codes listed above.

Fill Patterns

The following numbers may be used in functions requiring a fill pattern value.

Index	Pattern	Example
1	Solid	
2	Even color mix	
3	Dominant background color mix	
4	NW-SE diagonals	
5	NE-SW diagonals	
6	Vertical lines	
7	Horizontal lines	

8	Vertical squiggles	
9	Horizontal squiggles	
10	Vertical jagged lines	
11	Horizontal jagged lines	
12	Diagonal jagged lines	
13	Vertical crosshatch	
14	Vertical and angled crosshatch	
15	Angled crosshatch	
16	Uneven crosshatch	
17	Hollow connected squares	
18	Checkerboard	
19	Vertical arrows	
20	Horizontal arrows	
21	Angled checkerboard	
22	Large dots	
23	Thick connected squares	
24	Squares with tiny dots	
25	Bricks	

Font Character Sets

Returns the VTScada character set as an actual charset code. You may disregard these legacy presets and use the actual codes directly (see wingdi.h for list).

0	ANSI_CHARSET
1	DEFAULT_CHARSET (Current system charset. Typically, the same as ANSI_CHARSET)
2	SYMBOL_CHARSET
3	SHIFTJIS_CHARSET
4	OEM_CHARSET (System specific)
5	RUSSIAN_CHARSET
6	BALTIC_CHARSET
7	CHINESEBIG5_CHARSET
8	EASTEUROPE_CHARSET
9	GB2312_CHARSET
10	GREEK_CHARSET
11	HANGUL_CHARSET
12	MAC_CHARSET
13	TURKISH_CHARSET
14	VIETNAMESE_CHARSET

GUI Object Return Codes

Objects created using the GUIx commands will return a value when selected by the mouse or the enter button, according to the following table:

Return Value	Mouse Button(s)/Key	No. of Clicks
0	Invalid response	-
1	Right button	Single
2	Middle button	Single
3	Right and middle button	Single
4	Left button	Single

5	Left and right button	Single
6	Left and middle	Single
7	All three buttons	Single
8	Invalid response	-
9	Right button	Double
10	Middle button	Double
11	Right and middle button	Double
12	Left button	Double
13	Left and right button	Double
14	Left and middle	Double
15	All three buttons	Double
16	<ENTER> key	-

These are built from the following bit-wise values:

Bit	Meaning
0	TRUE: Right button clicked
1	TRUE: Middle button clicked
2	TRUE: Left button clicked
4	TRUE: Double-click, FALSE: Single-click
5	TRUE: Enter key presses (all other bits will be zero)

Known Path Aliases for File-Related Functions

Known Path Alias	Location
:{CommonProgramFiles}	Common program files (varies for 32-bit and 64-bit operating systems)
:{Fonts}	Fonts folder
:{ProgramFiles}	program files directory (varies for 32-bit and 64-bit

	operating systems)
:{System}	Windows System folder (varies for 32-bit and 64-bit operating systems)
:{SystemX86}	Windows System folder (32bit)
:{Windows}	Windows folder
:{ResourceDir}	Windows Resources folder

User Specific	
:{UserAdminTools}	Admin tools (start menu)
:{UserRoamingAppData}	Roaming app data
:{UserCDBurnArea}	Local pending CD to burn
:{UserCookies}	IE cookies
:{UserDesktop}	Desktop folder
:{UserFavorites}	IE favorites
:{UserHistory}	IE history
:{UserInternetCache}	Temporary internet files
:{UserLocalAppData}	Localized app data
:{UserDocuments}	Documents folder
:{UserMusic}	Music folder
:{UserPictures}	Photos folder
:{UserVideos}	Videos folder
:{UserNetHood}	Network Places shortcut folder
:{UserPrintHood}	Printer shortcut folder
:{UserProfile}	Users profile folder (root of users folders)
:{UserPrograms}	Programs (start menu)
:{UserRecent}	Shortcuts to recently viewed documents
:{UserSendTo}	Items in the 'Send To' context menu

:{UserStartMenu}	Start menu root
:{UserStartup}	Start menu startup folder
:{UserTemplates}	Document templates

All Users share the following:	
:{CommonAdminTools}	admin tools (start menu)
:{CommonAppData}	app data storage
:{CommonFavorites}	IE favorites folder
:{CommonOEMLinks}	OEM Links
:{CommonPrograms}	start menu program list
:{CommonStartMenu}	start menu root
:{CommonStartup}	start menu startup folder
:{CommonTemplates}	templates folder
:{PublicDesktop}	desktop folder (shared icons, etc…)
:{PublicDocuments}	documents folder
:{PublicMusic}	music folder
:{PublicPictures}	photos folder
:{PublicVideos}	video folder

Line Types

The following numbers may be used in functions requiring a line style value.

Index	Style	Example
0	Invisible	
1	Solid	———

2	Dashed	— — —
3	Dotted	----------
4	Dot-dashed	- — - — - ·
5	Dot-dot-dashed	--—·-·—·-

ParameterEdit Snap-ins

These are modules that do not stand alone, but can be used for parameter editing in the user interface.

ParmEditColor

Description Used for choosing a color as your parameter value.

Parameters
Color – Parameter value
ParmPtr – Parameter pointer
Enable – Show the graphics for editing the parameter value
Left – Left coordinate
Bottom – Bottom coordinate
Right – Right coordinate
Top – Top coordinate
LabelWidth – Label width
LabelHeight – Label height
PtrWaitClose – TRUE to tell caller to wait to close

ParmEditExprMovement

Description Used for choosing an expression as your parameter value for a movement parameter

Parameters
Expr – Parameter value
ParmPtr – Parameter pointer
Enable – Show the graphics for editing the parameter

value
Left – Left coordinate
Bottom – Bottom coordinate
Right – Right coordinate
Top – Top coordinate
LabelWidth – Label width
LabelHeight – Label height
PtrWaitClose – TRUE to tell caller to wait to close
DlgRoot – Root of the edit dialog (UNUSED)
DropListLabel0 – Label 0 for the Movement direction drop-list
DropListLabel1 – Label 1 for the Movement direction drop-list

ParmEditExprNoNormalize

Description Used for choosing an expression as your parameter. Note that, the result is not wrapped with a Normalize.

Parameters
Expr – Parameter value
ParmPtr – Parameter pointer
Enable – Show the graphics for editing the parameter value
Left – Left coordinate
Bottom – Bottom coordinate
Right – Right coordinate
Top – Top coordinate
LabelWidth – Label width
LabelHeight – Label height
PtrWaitClose – TRUE to tell caller to wait to close
DlgRoot – Root of the edit dialog (UNUSED)
MenuEnables – Bits to enable options for expression editor

ParmEditExprNormalize

Description ParameterEdit module for choosing an expression as your

parameter. The result is wrapped in a Normalize.

Parameters

Expr – Parameter value
ParmPtr – Parameter pointer
Enable – Show the graphics for editing the parameter value
Left – Left coordinate
Bottom – Bottom coordinate
Right – Right coordinate
Top – Top coordinate
LabelWidth – Label width
LabelHeight – Label height
PtrWaitClose – TRUE to tell caller to wait to close
DlgRoot – Root of the edit dialog (UNUSED)
MenuEnables – Bits to enable options for expression editor

ParmEditFont

Description

Used by the ParameterEdit to choosing a Font as your parameter value

Parameters

FontVal – Parameter value
ParmPtr – Parameter pointer
Enable – Show the graphics for editing the parameter value
Left – Left coordinate
Bottom – Bottom coordinate
Right – Right coordinate
Top – Top coordinate
LabelWidth – Label width
LabelHeight – Label weight
PtrWaitClose – TRUE to tell caller to wait to close
DlgRoot – Root of the edit dialog (UNUSED)

ParmEditHorizAlign

Description

ParameterEdit module for choosing horizontal alignment

as your parameter value.

Parameters

HAlign – Parameter value
ParmPtr – Parameter pointer
Enable – Show the graphics for editing the parameter value
Left – Left coordinate
Bottom – Bottom coordinate
Right – Right coordinate
Top – Top coordinate
LabelWidth – Label width
LabelHeight – Label weight
PtrWaitClose – TRUE to tell caller to wait to close
DlgRoot – Root of the edit dialog (UNUSED)

ParmEditLineStyle

Description

ParameterEdit module for choosing a line style as your parameter value

Parameters

Style – Parameter value
ParmPtr – Parameter pointer
Enable – Show the graphics for editing the parameter value
Left – Left coordinate
Bottom – Bottom coordinate
Right – Right coordinate
Top – Top coordinate
LabelWidth – Label width
LabelHeight – Label weight
PtrWaitClose – TRUE to tell caller to wait to close
DlgRoot – Root of the edit dialog (UNUSED)

ParmEditLineWidth

Description

ParameterEdit module for choosing a line width as your parameter value

Parameters

Width – Parameter value
ParmPtr – Parameter pointer
Enable – Show the graphics for editing the parameter value
Left – Left coordinate
Bottom – Bottom coordinate
Right – Right coordinate
Top – Top coordinate
LabelWidth – Label width
LabelHeight – Label weight
PtrWaitClose – TRUE to tell caller to wait to close
DlgRoot – Root of the edit dialog (UNUSED)

ParmEditNum

Description

ParameterEdit module for choosing a number as your parameter value

Parameters

NumVal – Parameter value
ParmPtr – Parameter pointer
Enable – Show the graphics for editing the parameter value
Left – Left coordinate
Bottom – Bottom coordinate
Right – Right coordinate
Top – Top coordinate
LabelWidth – Label width
LabelHeight – Label weight
PtrWaitClose – TRUE to tell caller to wait to close
DlgRoot – Root of the edit dialog (UNUSED)
TypeOfValue – Optional, VTScada value type required
MinVal – Optional, Min value allowed (Max also required)
MaxVal – Optional, Max value allowed (Min also required)

ParmEditParmMovement

Description

ParameterEdit module for choosing a container's para-

meter as a value. Used for displaying the Movement parameter. Has all the additional information needed for the Movement GUI call

Parameters

ParmVal - Parameter value
ParmPtr - Parameter pointer
Enable - Show the graphics for editing the parameter value
Left - Left coordinate
Bottom - Bottom coordinate
Right - Right coordinate
Top - Top coordinate
LabelWidth - Label width
LabelHeight - Label weight
PtrWaitClose - TRUE to tell caller to wait to close
DlgRoot - Root of the edit dialog (UNUSED)
ContainerInfo - Info struct for the container and its parameters
DropListLabel0 - Droplist label 0
DropListLabel1 - Droplist label 1

ParmEditParmValue

Description

ParameterEdit module for choosing a container's parameter as a value.

NOTE. Due to the introduction of PickValids for default parameters, this selector must appear in the modules array before any of the expressions.

Parameters

ParmVal - Parameter value
ParmPtr - Parameter pointer
Enable - Show the graphics for editing the parameter value
Left - Left coordinate
Bottom - Bottom coordinate
Right - Right coordinate
Top - Top coordinate

LabelWidth - Label width
LabelHeight - Label weight
PtrWaitClose - TRUE to tell caller to wait to close
DlgRoot - Root of the edit dialog (for sizing)
ContainerInfo - Info struct for container and its parameters
ValueTypeLo - Min value type allowed for this parameter. Note - If ValueTypeHi is invalid, then this is the only value type allowed.
ValueTypeHi - Max value type allowed for this parameter
SubTypeList - Optional list of subtypes for object type
Scaled - Optional Boolean to indicate whether the data be in a "Scale" expression. The default is FALSE
DefaultValue - Optional. The default value if the parameter's value is invalid
DefaultNoParm - Optional. The default value if no parameter is selected

ParmEditPattern

Description

ParameterEdit module for choosing a fill pattern as your parameter value

Parameters

Brush - Parameter value
ParmPtr - Parameter pointer
Enable - Show the graphics for editing the parameter value
Left - Left coordinate
Bottom - Bottom coordinate
Right - Right coordinate
Top - Top coordinate
LabelWidth - Label width
LabelHeight - Label weight
PtrWaitClose - TRUE to tell caller to wait to close
DlgRoot - Root of the edit dialog (UNUSED)

ParmEditPipeColor

Description ParameterEdit module for choosing a pipe color as your parameter value

Parameters

HighColor - Parameter value
ParmPtr - Parameter pointer
Enable - Show the graphics for editing the parameter value
Left - Left coordinate
Bottom - Bottom coordinate
Right - Right coordinate
Top - Top coordinate
LabelWidth - Label width
LabelHeight - Label weight
PtrWaitClose - TRUE to tell caller to wait to close
DlgRoot - Root of the edit dialog (UNUSED)

ParmEditPipeWidth

Description ParameterEdit module for choosing a pipe width as your parameter value

Parameters

PipeWidth - Parameter value
ParmPtr - Parameter pointer
Enable - Show the graphics for editing the parameter value
Left - Left coordinate
Bottom - Bottom coordinate
Right - Right coordinate
Top - Top coordinate
LabelWidth - Label width
LabelHeight - Label weight
PtrWaitClose - TRUE to tell caller to wait to close
DlgRoot - Root of the edit dialog (UNUSED)

ParmEditTag

Description Parameter Editing module for choosing a tag as your para-

meter value

Parameters

TagParm – Parameter value
ParmPtr – Parameter pointer
Enable – Show the graphics for editing the Parameter Value
Left – Left coordinate
Bottom – Bottom coordinate
Right – Right coordinate
Top – Top coordinate
LabelWidth – Label width
LabelHeight – Label weight
PtrWaitClose – TRUE to tell caller to wait to close
DlgRoot – Root of the edit dialog (UNUSED)
PointType – Point Type allowed for this parameter

ParmEditTagMovement

Description

Parameter Editing module for choosing a tag value as your parameter value for a Movement parameter

Parameters

TagMoveVal – Parameter value
ParmPtr – Parameter pointer
Enable – Show the graphics for editing the parameter value
Left – Left coordinate
Bottom – Bottom coordinate
Right – Right coordinate
Top – Top coordinate
LabelWidth – Label width
LabelHeight – Label height
PtrWaitClose – TRUE to tell caller to wait to close
DlgRoot – Root of the edit dialog (UNUSED)
DropListLabel0 – Label 0 for the movement direction drop-list
DropListLabel1 – Label 1 for the movement direction drop-list

PointType - Point Type allowed for this parameter

ParmEditTagProperty

Description

ParameterEdit module for choosing a property of the drawn tag.
NOTE. Due to the introduction of PickValids for default parameters, this selector must appear in the Modules array BEFORE any of the Expressions.

Parameters

ParmVal - Parameter value
ParmPtr - Parameter pointer
Enable - Show the graphics for editing the parameter value
Left - Left coordinate
Bottom - Bottom coordinate
Right - Right coordinate
Top - Top coordinate
LabelWidth - Label width
LabelHeight - Label height
PtrWaitClose - TRUE to tell caller to wait to close
DlgRoot - Root of the edit dialog (for sizing)
ContainerInfo - Info struct for container and its parameters
TargetTypeLo - Min value type allowed for this parameter. If TargetTypeHi is Invalid, then this is the only value type allowed.
TargetTypeHi - Max value type allowed for this parameter
TargetSubTypes - Optional list of subtypes for object type
Scaled - Should this value be scaled?
DefaultValue - Optional, default value if the tag's value is invalid
DefaultNoTag - Optional, default value if user selects 'No Property Selected'

ParmEditTPMovement

Description ParameterEdit module for choosing a container's parameter as a value

Parameters

ParmVal - Parameter value
ParmPtr - Parameter pointer
Enable - Show the graphics for editing the parameter value
Left - Left coordinate
Bottom - Bottom coordinate
Right - Right coordinate
Top - Top coordinate
LabelWidth - Label width
LabelHeight - Label height
PtrWaitClose - TRUE to tell caller to wait to close
DlgRoot - Root of the edit dialog (for sizing)
DropListLabel0 - Label 0 for the movement direction drop-list
DropListLabel1 - Label 1 for the movement direction drop-list
ContainerInfo - Info struct for container and its parameters

ParmEditTagValue

Description ParameterEdit module for choosing a tag value as your parameter value

Parameters

TagParm - Parameter value
ParmPtr - Parameter pointer
Enable - Show the graphics for editing the parameter value
Left - Left coordinate
Bottom - Bottom coordinate
Right - Right coordinate
Top - Top coordinate
LabelWidth - Label width

LabelHeight – Label height
PtrWaitClose – TRUE to tell caller to wait to close
DlgRoot – Root of the edit dialog (UNUSED)
PointType – Point type allowed for this Parameter
Scaled – Optional Boolean value indicating whether the data should be in a Scale expression. Default = true
DefaultValue – Optional, default value if the tag's value is invalid
DefaultNoTag – Optional, default value if no tag is set

ParmEditText

Description ParameterEdit module for choosing a text message as your parameter value

Parameters
TextMsg – Parameter value
ParmPtr – Parameter pointer
Enable – Show the graphics for editing the parameter value
Left – Left coordinate
Bottom – Bottom coordinate
Right – Right coordinate
Top – Top coordinate
LabelWidth – Label width
LabelHeight – Label height
PtrWaitClose – TRUE to tell caller to wait to close
DlgRoot – Root of the edit dialog (UNUSED)
StringLiteral – TRUE to treat input/output as a string literal intended for placement in SRC code or similar. (Handles quotation marks.) Defaults to TRUE.

ParmEditTextExpression

Description ParameterEdit module for choosing an expression that returns text as the parameter Value. Related to ParameterPanel, which calls the above.

Parameters

Expr - Parameter value
ParmPtr - Parameter pointer
Enable - Show the graphics for editing the parameter value
Left - Left coordinate
Bottom - Bottom coordinate
Right - Right coordinate
Top - Top coordinate
LabelWidth - Label width
LabelHeight - Label height
PtrWaitClose - TRUE to tell caller to wait to close
DlgRoot - Root of the edit dialog (UNUSED)
MenuEnables - Bits used to enable options for the expression editor

ParameterPanel

Description

This is a Generic Panel which will handle the setting of Parameters for an object that does not have its own panel. The object can be a page or a widget.
If available, hints are taken from the typing information of the object parameters and suitable Parameter Value choosers are offered.
If the immediate container has parameters then, these too are considered as actual value candidates. However, if there is type information on the object or the container, then these hints are used to filter the set of options, possibly resulting in an empty set.
This panel is designed to be callable from several sources, such as the VGE and the Display Manager.

Parameters

ObjModule - Object module that is to have its parameters modified
pObjParams - Pointer to an array of the object's parameter values
PtrWaitClose - TRUE when window closed or cancel

pressed

CodePtr - CodePtr so that the parameter value can be read

Left - Left position of the window

Bottom - Height of window

Right - Width of the window

Top - Top position of the window

HandleScrollBar - Flag - TRUE for this module to use a scrollbar if required

ContainerInfo - Information about container and its parameters

SelectedParms - Boolean array of selected parameters to make editable

DialogRoot - The calling dialog window

Related Functions:

ParameterEdit

SlippyMapRemoteTileSource1

Sets the URL, from which Site Map tiles are loaded. Defaults to: SlippyMapRemoteTileSource1 = http://c.tile.openstreetmap.org/ /|/ © [OpenStreetMap Contributors] (www.openstreetmap.org/copyright).

SQL Data Types

Type Indicator	SQL Data Type
-7	SQL_BIT
-6	SQL_TINYINT
-5	SQL_BIGINT
-4	SQL_LONGVARBINARY

-3	SQL_VARBINARY
-2	SQL_BINARY
-1	SQL_LONGVARCHAR
0	SQL_UNKNOWN_TYPE
1	SQL_CHAR
2	SQL_NUMERIC
3	SQL_DECIMAL
4	SQL_INTEGER
5	SQL_SMALLINT
6	SQL_FLOAT
7	SQL_REAL
8	SQL_DOUBLE
9	SQL_DATE
10	SQL_TIME
11	SQL_TIMESTAMP
12	SQL_VARCHAR

predefined Time Formats

Use any of the following numeric time codes to format a time as shown. If you require a custom format, you can build one using the text codes shown in Time Formatting Codes, as described in the VTScada Programmer's Guide.

Time Code	Example	Description
0		no time
1	103211	HourMinuteSecond

2	10:32:11	hour:minute:seconds
3	10321100	HourMinuteSecondHundredth
4	10:32:11:00	hour:minute:seconds:hundredths
5	10:32	hour:minute
6	10:32:11 AM	hour:minute:seconds AM or PM
7	10:32 AM	hour:minute AM or PM
8	103211000	hour minute second thousandths
9	10:32:11.000	hour:minute:second.thousandths

Related Information:

predefined Date Codes - Used to display a date value.

Time- Function that uses the codes listed above.

Time Formatting Codes

All examples display 9:07:12 p.m.

String	Example	Description
h	9	Hours with no leading zero for single-digit hours; 12-hour clock.
hh	09	Hours with leading zero for single-digit hours; 12-hour clock.
H	21	Hours with no leading zero for single-digit hours; 24-hour clock.
HH	21	Hours with leading zero for single-digit hours; 24-hour clock.

m	7	Minutes with no leading zero for single-digit minutes.
mm	07	Minutes with leading zero for single-digit minutes.
s	12	Seconds with no leading zero for single-digit seconds.
ss	12	Seconds with leading zero for single-digit seconds.
t	P	One character time-marker string, such as A or P.
tt	PM	Multi-character time-marker string, such as AM or PM.

Example:

```
Time(now(1), "hh.mm.ss tt")
```

... displays: 09.07.12 PM

Related Information:

Date Formatting Strings - Used to display a date value.
Time- Function that uses the codes listed above.

VTScada and Time Synchronization

Note: The following information is of concern only if you are not already maintaining time synchronization between servers and if it is important to your operation to maintain synchronization of clocks between servers.

VTScada includes code that will maintain time synchronization between networked servers. Due to security settings in Windows Vista and later versions, the Time Synchronization Service will work only if VTScada is started with an account that has the SE_SYSTEMTIME_NAME security flag set. The Windows Administrator account will generally have this privilege, but you can also set it for other user accounts.
To do so, you must set the user privilege, "Change the System Time" to true using the Windows Group Policy Management Editor. Please refer to Microsoft's documentation for instructions on using this system tool.

VTScada Value Types - Numeric Reference

The following table lists the value types used in VTScada. When referring to these in code, you should use the predefined constants rather than the type numbers. The general usage is:
Cast(Val, \#VtypeText)

Type	Constant Name	Name	Description
0	#VTypeStatus	Boolean	Logical data type, stores two states: "true" (0) or "false" (non-zero).
1	#VTypeShort	Short, 16-bit signed	Integer data type storing values from -32768 to 32767
2	#VTypeLong	Long, 32-bit signed	Integer data type storing values from -2147483648 to 2147483647
3	#VTypeDouble	Double precision floating point	Values range from about -10^308 through +10^308
4	#VTypeText	Text	Any string of bytes whose values range from 0 to 255. Typically used to hold text strings.
5	#VTypeVariable	Variable	A handle to the data represented by a variable declaration, not to any particular instantiation of that declaration. Can be used to access variable metadata (type information, for example) or default values.
6	#VTypeFunction	Function	A pointer to the code for a particular function within a VTScada statement. Used by functions such as GetOneParmText to

			manipulate the code itself. Used when compiling and editing script code, not for typical VTScada programs.
7	#VTypeObject	Object value	An instance of a module
8	#VTypeStream	Stream	A handle to a stream (of which there are several types). See Streams.
9	#VTypeModTree	Module tree	A handle to the modules in a state diagram
10	#VTypeStateDgrm	State diagram	A graphical depiction of VTScada code
11	#VTypeModule	Module	The code and variables that make up a unit of a VTScada program. See Modules.
12	#VTypeModState	Code Value (a) Module and state	A handle to a state within a module. See States.
13	#VTypeModStateStmnt	Code Value (b) Module, state, and statement	A handle to a statement within a state. Cannot refer to any arbitrary function, as type 6 can. See Statements and Graphic Objects.
14	#VTypeRefParm	Reference parameter	When a steady-state call is made to a module, each of the actual parameters in the call is "bound" to its corresponding formal parameter.
15	<undefined>	Array	Refers to an entire list of consecutive data values. Each data value has a consecutively numbered index address and may be any VTScada value. See Array Variables

16	#VTypePath	Path	A series of vertex values. See Path Variables.
17	#VTypeTraj	Trajectory	A combination of a Normalize value and a Path value. See Trajectory Variables.
18	#VTypeRotate	Rotate	Specifies a rotation amount, measured in degrees, around a point. See Rotate Variables
19	#VTypeBrush	Brush	Brush values are used in layered graphics statements that paint areas of the screen with a uniform color or pattern. See the Brush function.
20	#VTypePen	Pen	Pen values are used in layered graphics statements that draw lines. Defines the color, style and thickness of a line. See the Pen function.
21	#VTypeNormalize	Normalize	A graphical scaling value. See Normalize.
22	#VTypePoint	Point	A location, stored as an (X, Y) pair. See Point.
23	#VTypeVertex	Vertex	A group of three Point values. See Vertex.
24	#VTypeTransform	Transform	A transformation matrix, used to map coordinates from one area of the screen to another. Can only be obtained from the GetTransform function. Used by the GetPathBound function.
25	#VTypeCodePtr	Code pointer	A handle to an active graphics statement in a particular module

			or state. Similar to type 13, but with the additional information of the module instance as represented by value type 7.
26	#VTypePtr	Pointer	Stores data by reference instead of by value, allowing, for example, multiple values to reference the same piece of data as opposed to multiple copies of the data.
27	#VTypeEditor	Editor	A handle to an editor object, as created by MakeEditor.
28	#VTypeParseStack	Parser stack	Used by the compiler to allow the compilation to be suspended in the middle of a statement to handle specific code sections such as I/O addresses.
29	#VTypeTag	Tag	(Unused) Intended to provide engine-level support for scaled variables that could be implemented using a GUI.
30	#VTypeBitmap	Bitmap	A handle to an image object as returned from MakeBitmap.
31	#VTypeFont	Font	A handle to a font object, as returned by the Font function.
32	#VTypeVTSdb	VTScada database	A handle to the VTScada database as returned by the DBSystem function.
33	#VTypeODBCHndl	ODBC Handle	Provides a connection to an ODBC database.
34	#VTypeSAPIStrm	SAPI text-to-speech stream	A type of stream for use with Speech Application Programming Interfaces

35	#VTypeComClient	COM Client Interface	An object that provides an interface to a COM client application
36	#VTypeCryptoProv	Cryptographic Provider	A handle to the particular cryptographic service provider that includes the key specification to use.
37	#VTypeCryptoKey	Cryptographic Key	May be either a Session Keys or a Public/Private Key. See Cryptographic Keys.
38	#VTypeDLLhandle	DLL Handle	A pointer to a structure returned from the LoadDLL function. Used to call functions within the DLL that was loaded. See DLL.
39	#VTypeDeflateHandle	ZLib Compression Handle	Used by the Deflate function
40	#VTypeThread	Thread Handle	A script-level hook to the data structure used to represent a thread in a dump
41	#VTypeBreakWatch	Source Debugger Breakpoint Handle	References a set location in the source debugger. See Working with Breakpoints and Data Breakpoints
42	#VTypeMiniDumpHandle	Minidump Data Handle	A pointer to a data structure that holds information from a crash dump
43	#VTypeTimeStamp	Timestamp	A numeric representation of time, measured in seconds since January 1, 1970
44	#VTypeXMLproc	XML Processor Handle	Serves as a conduit between an XML document and an application. See VTScada Engine XML API

45	#VTypeTypeDefinition	Dynamic Module Definition	Deprecated. A handle to the definition of a form of module used as a data container. Created by the MakeType function. This storage is used almost exclusively for handling XML and cannot contain script code (unlike other forms of Module).
46	#VTypeTypeInstance	Dynamic Module Instance	Deprecated. An instance of a dynamic module, created using the MakeTypeInstance function. This is an object value (type 7) that can only be used to store data - it cannot contain or execute script. Typically these are used when generating module trees for delivery via XML. It is a form of data container, however in general structures (defined by the Struct function) and Dictionaries (type 47) are more efficient and convenient for this role.
47	#VTypeDictionary	Dictionary	A key-based data container of flexible size, used either on its own to hold volatile data collections or in the definition of structures (see Structures). ValueType will not return this value unless the dictionary is a "pure" dictionary. A pure dictionary is one for which the root value has not been set. Otherwise, it returns the ValueType of

			the dictionary's root instead. See Dictionaries
48	#VTypeComProperty	COM Property	A value exposed by a COM Interface "object". This may be accessed similarly to a typical VTScada value but is maintained by the COM object, not the VTScada engine.
49	<undefined>	Module in Context	Contains both a module value and an instance of the context module where scope should be resolved. Normally, scope will be the parent module in which the Module was declared. A Module in Context is used for widgets and plug-ins in VTScada where the widget is declared in AppRoot.SRC, but linked into a tag type such that the widget becomes a Module in Context in the tag instance. References to variables in the widget will then refer to variables in the tag rather than to variables in AppRoot where the widget was declared. If a Module In Context value is called in steady-state, the parent instance will provide the associated context.
50	#VTypeHistorianHandle	Historian Connection Handle	For the VTScada proprietary data store, this will be invalidated on an "out of disk space" error, or

			on loss of access to the file storage. For other databases, this will be invalidated on any connection loss.
51	#VTypeXMLNode	Dictionary Structure	A WEB_XML_ADDRESS that points to a WEB_XML_NODE. When ValueType() runs against a value and finds a WEB_XML_ADDRESS it treats it the same as a WEB_VALUE_ADDRESS, which sits in front of an array or structure. It then searches through the *_ADDRESS to find what it points to and returns the type of that item, in this case an USER_XML_NODE
52	#VTypePPPHandle	PPP Connection Handle.	May be passed into the function, PPPStatus() to obtain an information structure. May be passed to the function, CloseStream() to forcibly close off a connection. Passing it into CloseStream completely invalidates the handle and all data associated with it.(see: PPPStatus and CloseStream)

Value and Type Conversions

The following table shows VTScada value and type conversions.

Value to Convert	Convert To	Condition of Original Value	Returned Value

Code Pointer	Module	Valid value	Valid value
	Module State	Valid value	Valid value
	Module State		
	Statement	Valid value	Valid value
	Object	Valid value	Valid value
	Text	Valid value	Name of module
Double	Long	Valid value	Valid value
	Short	Valid value	Valid value
	Status	Valid value	Valid value
	Text	Valid value	String representing numeric value
Edit Block	Stream	May only be used in SRead with % option, or in StrLen	Valid value
Long	Double	Valid value	Valid value
	Short	Valid value	Valid value
	Status	Valid value	Valid value
	Text	Valid value	String representing numeric value
Module	Text	Valid value	Name of module
Module State	Module	Valid value	Valid value
	Text	Valid value	Name of module
Module State Statement	Module	Valid value	Valid value
	Module State	Valid value	Valid value
	Text	Valid value	Name of module
Normalize	Double	Valid value	The scaled value
	Long	In the range of -2 147 483 648 to 2 147 483 647	The value

	Short	In the range of -32 767 to 32 767	The value
	Outside of range	Invalid	
	Status	0	0 (false)
		Non-0	1 (true)
	Text	Valid value	String representing scaled value
Object	Module State	Valid value	Module and state in which that object exists
	Statement	Valid value	Module state and statement number that object is executing
	Text	Valid value	Name of the module of which that object is an instance
Short	Double	Valid value	Valid value
	Long	Valid value	Valid value
	Status	Valid value	Valid value
	Text	Valid value	String representing numeric value
Status	Double	Unconnected socket	1
		Connected socket	Invalid
	Long	Unconnected socket	1
		Connected socket	Invalid
	Short	Unconnected socket	1
		Connected socket	Invalid
	Status	Unconnected socket	1
		Connected socket	Invalid
	Text	Unconnected socket	"1"
		Connected socket	String of stream contents
Tag	Double	Valid value	The scaled value
	Long	In the range of -2 147 483 648 to 2 147 483 647	The value

		Outside the range	Invalid
	Short	In the range of -32 767 to 32 767	The value
		Outside the range	Invalid
	Status	0	0 (false)
		Non-0	1 (true)
	Text	Valid value	String representing scaled value
Text	Double	Number string	Number in string
	Long	Number string	Number in string
	Short	Number string	Number in string
	Status	Non-0 number string	0 (false)
		0 number string	1 (true)
Variable	Module	Module variable	The module in which the module variable resides
	Text	Valid value	Name of variable

Uninstall VTScada

Your VTScada installation includes a wizard that can assist you in uninstalling the entire VTScada suite, or selected VTScada components. Uninstalling VTScada is a simple, two-step process that removes all components of the VTScada software from your workstation. The uninstall process does not, however, affect any VTScada applications you've created. These applications and their resources will remain untouched. Follow these steps to completely uninstall VTScada from your system.

1. Ensure that VTScada is not running.
2. Navigate to the VTScada product or installation directory.
3. Locate the Uninstall.exe file and run it.
4. Ensure that the Automatic radio button is selected.
5. Click the Next button. The Perform Uninstall dialog opens.

6. Click the Finish button. The Uninstall Wizard removes all VTScada components from your workstation.

Note: Although the Uninstall Wizard removes all VTScada components from your workstation, it leaves any applications you have created and their data untouched. These applications can be imported into other versions of VTScada at a later date.

Language Support

The VTScada has been created using only English. Some developers have created application that present a portion of the user interface in languages other than English. To do so:

To use another language for your application's user interface:

1. Ensure that you are using fonts that support the full range of characters (including accented characters) used by the desired language. Modify the built-in the Font tags to use those fonts.
2. Ensure that the default Windows system font uses the correct selection for your language.
3. Edit your pages and pop-up pages to use the appropriate words for your language. Page titles can be edited as required using page properties.

To use another language for the VTScada configuration dialogs and messages:

1. Open the file, C:\VTScada\Setup.INI and replace the label properties with the appropriate words for your language.
   Note: do not edit the property names. Replace only the values attached to those names.
2. Re-start VTScada in order to load the new labels. Setup.INI is read only when VTScada restarts.
3. Using the advanced mode of the Edit Properties page of your application's Application Configuration dialog, replace label values with the appropriate

words for your language.
You will need to copy many of the properties from the OEM layer.

Related Information:

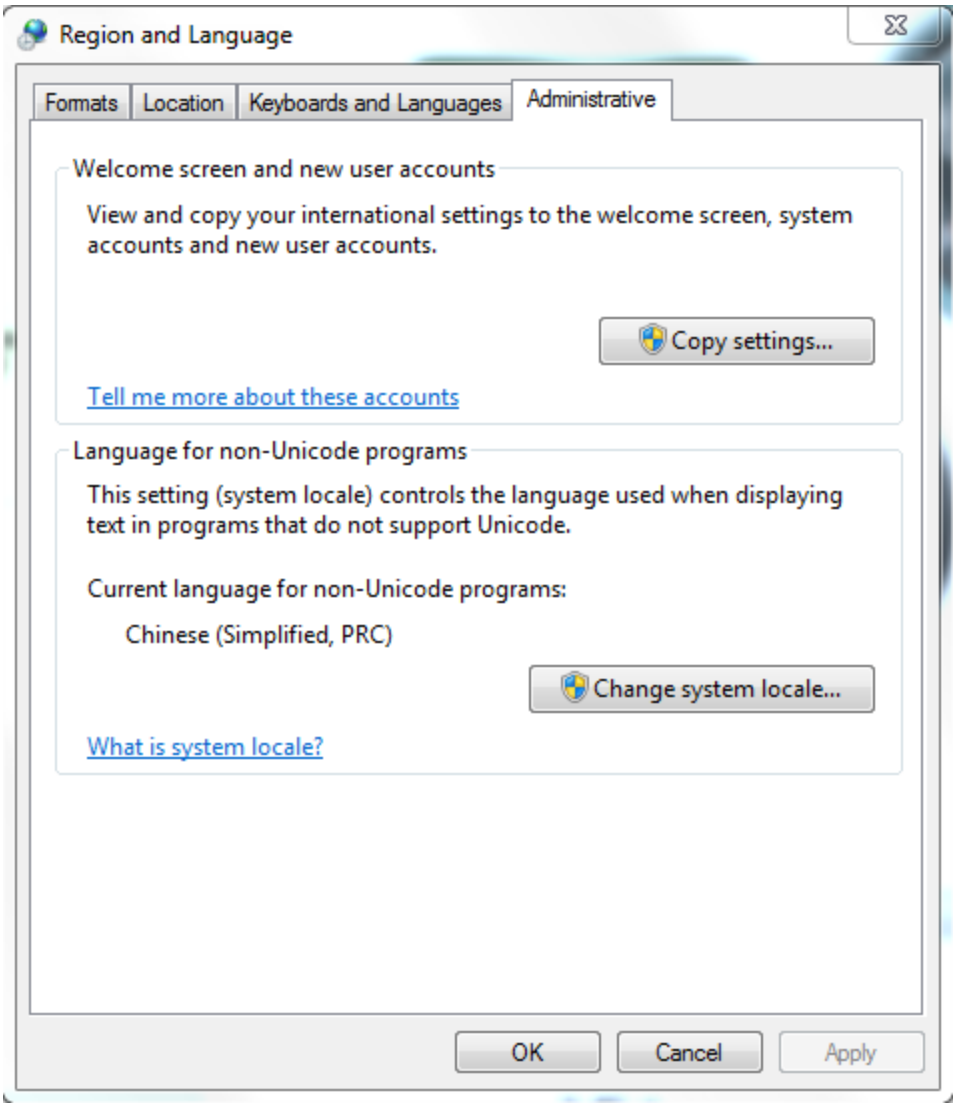
...Using a Non-English Character Set - Configure Windows (and thereby VTScada) to use alternative character sets.

Using a Non-English Character Set

To display non-English characters in VTScada, you may to do the following: (Steps describe how to use a Chinese character set.)

1. Update your Windows to include Chinese character set if you haven't done that when you install your Windows.
2. Go to Control Panel -> Clock, Language, and Region -> Region and Language Setting.
3. Select 'Administrative' tab and click on the "Change system Locale..." button.
4. If the client is from Main land China, select "Chinese (Simplified, PRC)" from the droplist.

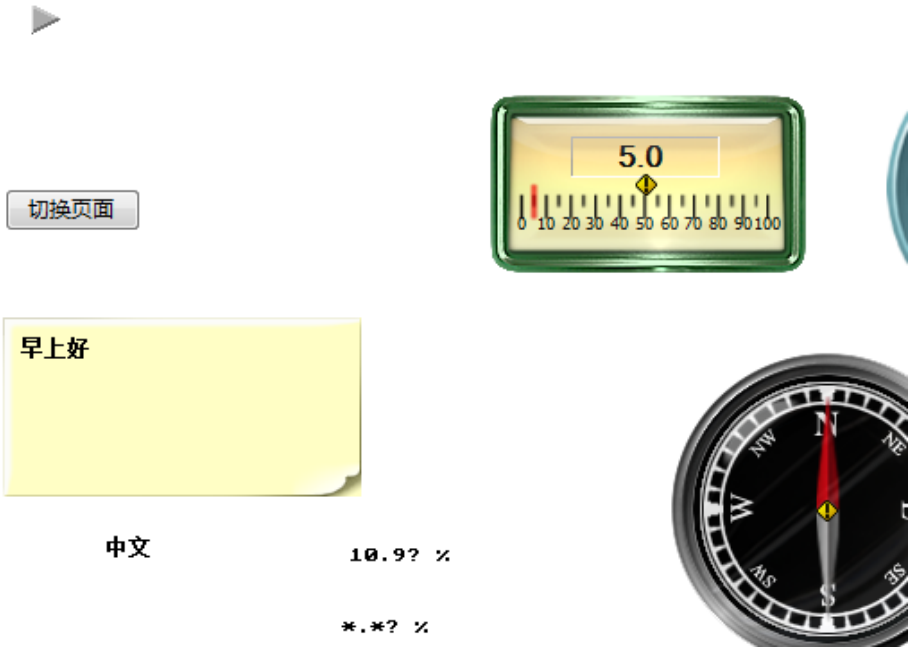
If the client is from Hong Kong, select Chinese (Traditional, Hong Kong S.A.R) and so on. The screen should look like the following after your selection.



Windows will ask you to restart your system.

5. If you want to input Chinese characters in VTScada, you need to go to Control Panel -> Region and Language setting, select 'Keyboards and Languages" tab and click on "Change keyboards..." button.
6. In the "Text Services and Input Languages" dialog, Add Chinese input in the "Installed services" and keep English as Default Input Language.
7. Run VTScada and you can edit text messages, button labels or Page Notes in Chinese.

Note: tag names do not support Chinese characters.



Related Information:

...Language Support - Creating a user-interface with a language other than English.

VTScada Functions - Grouped by Type

Alarm Functions

Commission	(Alarm Manager module) Commission the alarm by adding it to the Configured list.
	(Alarm Manager module) Returns a deep copy of an alarm record.
Decommission	(Alarm Manager module) Decommission an alarm by name.
EvaluateAlarm	(Alarm Manager module) Passes a new value to an alarm, to be compared to the setpoint.
GetAlarmConfiguration	(Alarm Manager module) Returns a copy of an alarm's configuration structure from the master alarm list, creating it if it does not already exist.
GetAlarmList	(Alarm Manager module) Returns filtered lists of records from alarm databases.
GetAlarmObject	(Alarm Manager module) Returns an alarm object value given an alarm name.
GetAlarmStateStats	(Alarm Manager module) Returns a structure containing the cumulative alarm state statistics for the specified tag.
GetAlarmStatus	(Alarm Manager module) Returns a reference to an alarm's status structure in the master database, providing access the alarm's current state without having to make additional function calls.
GetContainerNumActive	Returns the number of active alarms within a hierarchy of tags.
GetContainerNumUnacked	Returns the number of unacknowledged alarms within a hierarchy of tags.
GetNameOfRecord	(Alarm Manager module) Given an alarm record,

	returns the tag name.
GetNumUnacked	(Alarm Manager module) Returns either the number of unacknowledged alarms or a Yes/No flag to indicate that the alarms exist. If realm-area filtering is in use, results will be limited to the operator's realm.
GetUserNameOfRecord	(Alarm Manager module) Given an alarm record, returns the user name associated with the trans-action.
IsActive	(Alarm Manager module) Will indicate if an alarm is currently active. GetAlarmStatus should be used in new code.
IsDisabled	(Alarm Manager module) Will indicate if an alarm is currently disabled. GetAlarmStatus should be used in new code.
IsShelved	(Alarm Manager module) Will indicate if an alarm is currently shelved. It can be used either as a sub-routine or as a called function.
IsUnacked	(Alarm Manager module) Will indicate if an alarm is currently unacknowledged. GetAlarmStatus should be used in new code.
MuteSound	(Alarm Manager module) This subroutine is used to turn off alarms sounds for all alarms, both current and future.
PAlmPriority	(VTS Library) Draws a droplist of the currently available alarm priorities with an optional title or bevel or both.
SetShelved	AlarmManager plug-in, that handles the shelving and unshelving of alarms.
SilenceSound	(Alarm Manager module) This subroutine will

	silence the current sounding alarm.
The following alarm functions are deprecated or obsolete as of version 11.2	
Acknowledge	*Deprecated.* (Alarm Manager module) Will acknowledge an alarm.
Active	*Deprecated.* (Alarm Manager module) Tells the Alarm Manager to activate an alarm. This sub-routine will activate an alarm and signal it as unacknowledged.
ActiveMonitor	*Obsolete.* (Alarm Manager module) Starts the required actions for an active alarm. Act-iveMonitor waits for an alarm to activate, then activates the Sounder and, (if enabled) calls AlarmManager to add the pop-up.
AlarmSoundCheck	*Obsolete.* (Alarm Manager module) Checks to see if an alarm sound can be played for the given alarm
Disable	*Deprecated.* (Alarm Manager module) Tell the Alarm Manager to disable an alarm. Disable will also clear any active or unacknowledged state that might exist.
DoAcknowledge	*Obsolete.* (Alarm Manager module) This sub-routine will do the actual work of acknow-ledging, while the Acknowledge subroutine is responsible for informing other computers in the network about the alarm acknowledgment.
Enable	*Deprecated* (Alarm Manager module) Tell the Alarm Manager to enable an alarm.
Event	*Deprecated.* (Alarm Manager module) Tell the Alarm Manager when an alarm event occurs.

	This subroutine will cause an entry to be added to the log file without changing the alarm status.
ListAdd	*Deprecated.* (Alarm Manager module) This subroutine will add the alarm object to a list. This is useful if a user-defined list has been created.
ListRemove	*Deprecated.* (Alarm Manager module) This subroutine will remove the alarm object from a list. This is useful if a user-defined list has been added.
Normal	*Deprecated.* (Alarm Manager module) Use this function to tell the Alarm Manager when an alarm clears. This subroutine will deactivate the alarm. It will not affect the unacknowledged status.
NormalTrip	*Deprecated* (Alarm Manager module) This subroutine will deactivate an alarm and signal it as unacknowledged.
OffNormal	*Deprecated.* (Alarm Manager module) This subroutine will activate the alarm. However, it will not affect the unacknowledged status.
Popup	*Deprecated.* (Alarm Manager module) Causes an alarm pop-up dialog to be displayed.
Register (Alarm Manager)	*Deprecated.* (Alarm Manager module) Inform the Alarm Manager that a module instance may wish to generate alarms in the future.
SetEnable	*Deprecated.* (Alarm Manager module) Use in preference to Enable to enable or disable alarms build into tags.

ShelvedEvent	*Obsolete.* AlarmManager plug-in, called to record a shelved alarm event.
StartSound	*Obsolete.* (Alarm Manager module) This subroutine is used to start a sound for an alarm if alarm sound is enabled and the alarm priority is higher than currently sounding alarm.
TransferFields	*Deprecated* (Alarm Manager module) The TransferFields subroutine will transfer the values for each field into the returned FieldValues array. The values are found in the scope passed in using the variable names found in the FieldNames array.
Trip	*Deprecated.* (Alarm Manager module) Tell the Alarm Manager to when a trip alarm event occurs. This subroutine will signal an alarm as unacknowledged.
Unregister (Alarm Manager)	*Deprecated.* (Alarm Manager module) Notify the Alarm Manager that an alarm has been removed. This will not generate an alarm; it just removes it from the list of all configured alarms.

Array

AdjustArray	Changes the array information for a variable.
AMax	Array maximum. This function returns the maximum value in a sub-range of a numeric array.
AMin	Array minimum. This function returns the minimum value in a sub-range of a numeric array.
ArrayDimensions	Returns the number of dimensions in an array.
ArrayOp1	Performs a mathematical operation on an array with respect to a scalar value.

ArrayOp2	Performs a mathematical operation on an array with respect to another array.
ArraySize	Returns the number of elements in an array dimension.
ArrayStart	Returns the first element in an array dimension
ArrayToBuff	Returns a buffer containing the numeric data from an array.
AValid	Returns the number of valid elements in an array sub-range.
BuffToArray	Reads an array from a formatted buffer containing numerical data and returns the number of elements read.
BuffToPointer	Converts a buffer of numeric data to array of pointers. This function reads from a formatted buffer containing numeric data, writes to locations specified by an array of pointers, and returns the number of elements read.
Compress	Eliminate invalid array entries.
DeleteArrayItem	Deletes an element from a single dimension dynamically allocated array and returns the modified array.
Filter	Sets the value of one array element to invalid if the corresponding value in another array element is invalid.
FiltHigh	Sets the values in an array sub-range that fall above a specified upper limit to a new value.
FiltLow	Sets the values in an array sub-range that fall below a specified lower limit to a new value.
InsertArrayItem	Insert Array Item. This function inserts an element into a dynamically allocated array and returns the modified array.
LookUp	Looks up a value in an array and returns the index of the element containing that value.
Mean	Returns the mean (average) of a portion of a numerical array.
New	Allocates memory for an array from RAM and returns a pointer to that array.
PointerToBuff	Returns a buffer containing the numeric data from the variables pointed at by each element of the array.

ReadX	Reads numeric data from a text file into the elements of an array.
ReadXY	Reads data points from a file into the elements of two arrays.
SaveHistory	This threaded function saves an array of data to a .DAT file for a certain time span.
SDev	Returns the statistical sample standard deviation for a sub-section of an array.
Sort	Allows the sorting of an array subsection according to the order of another array.
SortArray	Sorts an array of arrays based upon the key information provided by the second parameter.
Sum	Returns the arithmetic sum of all the valid array elements in a specified portion of a numeric array.
TextSearch	Returns the array index of the first occurrence of the given text key in an alphabetically ordered array.
Unpack	Unpacks a set of values from a stream into a single dimensional array or a set of variables referenced by object parameters, and returns the number of items unpacked.
UnpackData	(RPC Manager Library) This method unpacks a stream into an array or set of module instance parameters. Subroutine call only.
Variance	Returns the statistical sample variance for a subsection of an array.
WatchArray	Watches an array and returns true if any of its elements types or values change.

Bitwise Operation

And	Returns the bit-wise AND of its two parameters as a 32-bit unsigned integer.
Bit	Returns the on/off status of a bit in a number.
Not	Returns the result of a 32 bit unsigned bitwise logical NOT operation.

Ones	Returns the number of bits set in an integer number.
Or	Performs a bit-wise OR operation and returns the result.
PSecBit	(VTS Library) Parameter Setting Security Bit. This module draws a titled, beveled droplist of options for setting the security bit.
SetBit	Sets or clears a specific bit in a value and returns the result.
XOr	Returns the bitwise exclusive OR of its parameters.

Clipboard

ClipboardGet	Returns the current contents of the system clipboard as a string. This function enables an application to perform text "paste" operations.
ClipboardPut	Set the current contents of the system clipboard to a string. This function enables an application to perform text "copy" or "cut" operations.

Color

Blend	Returns an aRGB color value that is a given percentage between two specified colors.
Brush	Returns a brush value.
Cls	Clears the screen and sets its background color.
ColorSelect	(System Library) Color Selection Tool. This module draws a color selection button and its accompanying display area.
GetColorInfo	Returns the brush and pen information for a given graphic statement.
GetSystemColor	Returns the colors for the user-configured Windows™ colors.
PalStatus	Returns the current palette settings.
PColorSelect	(VTS Library) Draws a button that opens a color selection dialog and an area that displays the currently selected color.
Pen	Returns a pen value.
PixelColor	Returns the color of a pixel in the window.
ZColorChange	Changes one color within a region to another color.

Com

ActiveX	Instantiates an ActiveX object. An ActiveX object is treated as a COM client interface that requires a client window area in which to draw.
COMClient	Instantiates COM objects that do not possess a user interface.
COMEvent	Sets an event subroutine context for an existing COM client interface.
COMStatus	Returns the last status information that occurred for a specified COM client interface.

Compilation And On-Line Modifications

ActiveCode	ActiveCode returns the code value of the currently active statement in the given module instance.
ActiveState	ActiveState returns the code value of the currently active state in the given module instance.
ActiveWindow	ActiveWindow returns the object value of the root module instance in the current active window.
AddModule	Adds a new module to an existing module and returns the value of the newly created module.
AddOptional	Adds a new statement to an action script and returns its own error code.
AddPageToApp (Obsolete)	Creates a new application page.
AddParameter	Adds an existing variable as a module parameter and returns the number of parameters in the module.
AddState	Adds a new state to an existing module and returns its state value.
AddStatement	Adds a new statement to an existing state and returns its own error code.
AddVariable	Adds a new variable to an existing module and returns its variable value.
AdjustArray	Changes the array information for a variable.

AdjustCode	Adjusts the offsets and sizes of items stored in the .RUN file within the document file.
BuffToParm	Convert buffer of numeric data to parameters. This function reads module parameters from a formatted buffer containing numerical data and returns the number of data items read.
Call	Starts an instance of the module specified by its first parameter.
CalledInstances	Returns the object values of module instances that are called by a particular module.
Caller	Takes a given object value for a module and returns the object value of the module by which it was called.
CanEditDoc	Returns an indication as to whether or not the document for the given module can be edited.
Cast	Takes a value and returns a different type of value, if possible.
ChangePersistentSize	Changes the space allocated in the persistent value (.VAL) file for a variable.
ChildDocs	Gets the module values for the root and all descendent modules that match the conditions defined by the second parameter.
ClearModule	Deletes the contents (all variables and states) of a module without removing the module itself.
ClearState	Deletes all of the statements in a state.
Compile	Compiles text and creates a new function; its type of return value is determined by its input parameters.
ConstCount	Returns the number of constant parameters in a function.
CreateModule	Creates a new module and returns a pointer to it.
CriticalSection	Marks a section of a module as a critical section and will not allow interruption of its execution by other threads.

Debugger	(System Library) Starts the VTScada debugger.
DeleteModule	Deletes a module from the system.
DeleteOptional	Deletes a statement from an action script.
DeleteState	Deletes a state from a module.
DeleteStatement	Deletes a statement from a state.
DeleteVariable	Deletes a variable from a module.
DelPageFromApp	Deletes a system page from an application.
FileRootModule	Parses the document file that contains the given module to find the root module in that file. Returns the module value of the root module.
FindAction	Returns an action from the list of actions in a state.
FindVariable	Searches for a variable by text name and returns a variable value.
FirstState	Sets the first state in a module.
ForceState	Sets the next state to start when the action script completes.
FormalParms	Returns the number of formal parameters declared in a module.
GetDefaultValue	Returns a variable's default value.
GetID	This returns the ID (opcode) of a given function.
GetInstance	Returns the object value of a module instance.
GetModuleRefBox	Get Module Reference Box
GetModuleText	Returns information about a module's document file.
GetOneParmText	Returns the text for one parameter of a function.
GetOverrides	Returns an array of OpCodes and the module value that will run when each OpCode is executed
GetParmText	Returns the text for all parameters of a function.
GetParserOffset	Returns the offset before the last compiled statement.

GetReturnValue	Returns a module's return value.
GetState	Returns the code value for the specified state.
GetStatement	Returns the code value for the specified statement.
GetStatementNum	Returns the statement number for the specified statement.
GetStateText	Returns the text for the specified state.
GetToken	Reads the next token from a stream and returns the token type.
GetTransitText	Get Transition Document Text. This function returns information about the documentation of an action.
GetVariableText	Returns information about the documentation of a variable.
GetXformRefBox	Get Transform Reference Box. This function returns the reference box for any transform of a module.
LastSelected	Returns the most recently selected graphics statement.
ListVars	Returns a list of variables.
LoadDLL	Loads a Microsoft Windows™ dynamic link library.
LoadModule	Loads a module from its .RUN files and returns a pointer to that module.
LValue	Left-hand Side Value. This function returns an indication of whether its argument can be used on the left-hand side of an assignment.
MakeDAG	Constructs a Directed Acyclic Graph (DAG - an internal function representation).
MakeNonPersistent	Takes a variable and makes it not persistent.
MakeNonShared	Takes a shared variable and makes it not shared.
MakePersistent	Takes a variable and makes it persistent (static).
MakeShared	Takes a variable and makes it shared.
MCSInstance	Module Calling Structure Instance. This function returns the object value of a module called by another module.

MCSMod	Module Calling Structure Module. This function returns the module value from a line of code that calls that particular module.
ModuleFileName	Returns the full path (including the drive letter) and file name of the document (.SRC) file of a module.
NParm	Returns the number of parameters listed in a module instance.
NumParms	Returns the number of parameters of a statement.
NumSets	Returns the number of statements that are currently active in setting a particular variable.
NumVariables	Returns the number of variables in a module.
OwningModule	Returns the module which contains a certain variable.
ParserSRO	Adds a scope resolution reference to a variable on the top of the PARSER_STACK given the stack and the object variable.
PersistentSize	Returns the size in bytes of a variable's persistent value size in the persistent value (.VAL) file.
RemoveParameter	Removes a parameter from a module's parameter list.
ReplaceStatement	Replaces a statement with another statement.
ResetParm	Can reset parameters that become latched.
ResyncDoc	Synchronizes the time and date for the document and .RUN files.
RUNFileName	Returns the name of the .RUN file for a module including the full drive and path.
RUNFileVersion	Returns the minimum version of VTScada that can read the .RUN files produced by the current version.
SaveModule	Saves a module definition to its *.RUN file.
SetDefault	Sets the default value for a variable.
SetModuleRefBox	Sets the reference box for a single instance of a module.

SetLibrary	Sets the library for an application.
SetModuleRefBox	Sets the default reference box for a module.
SetModuleText	Sets the module's .SRC file information.
SetOneParmText	Sets the text for one parameter of a function.
SetParameter	Sets a parameter in a statement.
SetParmText	Sets the text for the parameters of a function.
SetParserParm	Sets the value for the last parameter on the parser stack and returns its own error code.
SetRefRect	Sets the first four constant parameters of a layered graphic statement.
SetStateText	Sets the information about the text of a state in a .SRC file.
SetTransfer	Sets the destination for an action.
SetTransitText	Sets the information about the documentation of an action in the .SRC file.
SetVariableClass	Sets the class number of a variable and returns its previous class number.
SetVariableText	Sets the information about the documentation of a variable in the .SRC file.
SetVariableType	Sets the data type for the variable, so that only values of that data type can be stored in the variable.
StateList	Returns a list of states for a module.
StatementInstance	Takes a given code value and object and returns a code pointer value for that instance.
StateName	Returns the text name of the given state.
SubStatementIndex	Returns the index of a function within the statement where it is called.
VarAttributes	Returns the attributes bit field of a variable.

Configuration

ModifyTags	Can be used to create, modify, or delete running tags. Replacement for StartTag.
OpChange	Wrapper for TagMigrator\OpChange. Performs an immediate deploy of a single tag change without disturbing any other tag changes already in place on the local branch.
SimpleOpChange	Immediately deploys a single parameter change on a single tag without disturbing any other tag. Makes use of OpChange.
StartTag	(Deprecated. See: ModifyTags) Is used to create tags by starting new instances of the tag type specified in the parameter list. When creating an application that requires child tags, it is recommended that this function be used in place of the older ChildLaunch function.

Configuration Management

AcquireLock	Subroutine to acquire an exclusive lock on reading/writing working copy files across all applications.
AppIsRunning	Reports whether the application has been started and the start-up process is complete.
AppIsStarted	Returns TRUE if the application has been started.
AppIsStarting	Returns TRUE if the application is currently in the process of starting.
ApplyChangeSetFile	Apply a named ChangeSet to an application layer.
CaptureSettings	Gathers a single property value or an accumulated section and returns the result in a tabular format.
Combine	Performs a Merge2 operation with automated conflict resolution and change priority.
CommitEditedFiles	This function compiles and commits edited files if the compile succeeds.
DirectApply	Applies a set of changes directly to the repository, without disturbing existing (non-conflicting) changes already on either branch.
EditFile	Informs the configuration management system that a

	file has been modified in the working copy, typically before making a call to CommitEditedFiles.
GetAppInstance	Asynchronously, retrieves the Layer object (LayerRoot) for a particular application specified by its GUID.
GetCodeObj	Retrieves the "Code" object associated with the layer.
GetINIProperty	Given an array of INIProperty structures, returns the value of a given property from that array.
GetLoadedAppInstance	Synchronously, retrieves the Layer object (LayerRoot) for a particular application specified by its GUID.
GetOEMLayer	Retrieves the layer root module of the OEM layer (should one exist) of the layer this is called against.
GetPlatformInfo	Gathers information about the current application and the workstation it is running on.
GetWCPath	Returns the full working copy path for an application.
GetWCRevision	Returns the revision structure for the repository revision currently in use by the working copy.
HasCompilationErrors	Reports if the working copy presently has unresolved compilation errors
HasUndeployedChanges	Finds whether the local machine is maintaining changes that have not been deployed, including changes that have been recorded by EditFile but have yet to be committed.
IsAppEditable	Returns TRUE if the application can currently accept changes without being re-started.
IsOnLocalBranch	Returns TRUE if the local machine is maintaining changes that have not been deployed within the repository.
IsRunOnly	Returns TRUE if the application is a run-file-only app, according to the WC contents.
LayerInUse	Returns true if the application is running, or if there are any applications that depend on this layer, running or

	not.
Merge	Applies a set of changes (the output of a Diff operation) to a buffer.
Merge2	Attempts to apply two different Diff buffers to a single origin buffer.
ReadINIProperties	Gathers the sum of all of the properties files in this layer and all of its parents including the local work-station files.
ReleaseLock	Releases a working copy semaphore that was acquired by AcquireLock.
ReadPropertiesFile	Reads a single Settings file and returns an INIFile Struc-ture.
RecordProperty	Helper function used to record settings without need-ing to explicitly interact with the settings files.
RepoSubscribe	Allows the caller to specify a callback which will be triggered whenever the application's repository changes.
SetINIProperty	Given an INIFiles structure, this function sets the prop-erty with the specified name and section to the spe-cified value
Start	Start an application.
LayerRoot\Stop	Stop an application.
WriteINIProperties	The opposite to ReadINIProperties.
WritePropertiesFile	Write a single Settings file according to the properties in an INIFile structure.

Container And Contributor

AddContributor	Adds a contributor to a container.
DeleteContributor	Removes a contributor from a container.
GetContributors	Returns a copy of an array of object values of contributors for a given container.

PContributor	(VTS Library) Draws a splitlist displaying all contributors to a specific tag.

Cryptography

Base64Decode	Performs a Base64 decode of a buffer.
Base64Encode	Performs a Base64 encode of a buffer
Decode	Returns the plain value of a cipher that is the result of the Encode function.
Decrypt	The Decrypt function decrypts data previously encrypted using the Encrypt function. It is the VTScada analog of the CryptoAPI CryptDecrypt call.
DeriveKey	Generates a cryptographic session key from a seed value.
Encode	Processes a VTScada string using a configurable selection of compression, encryption, encoding and secure hashing.
Encrypt	The Encrypt function encrypts data. The algorithm used to encrypt the data is designated by the Key parameter. It is the VTScada analog of the CryptoAPI CryptEncrypt call.
ExportKey	The ExportKey function exports a cryptographic key or a key pair from a CSP in a secure manner as a Key BLOB. It is the VTScada analog of the Crypto API ExportKey call.
GenerateKey	The GenerateKey function generates a random cryptographic session key or a public/private key pair. A handle to the key or key pair is returned. This handle can then be used as needed with any CryptoAPI function requiring a key handle. It is the VTScada analog of the CryptoAPI's CryptGenKey call.
GetCryptoProvider	The GetCryptoProvider function is used to acquire a handle to a particular key container within a particular cryptographic service provider (CSP). This returned handle can then be used to make calls to the selected CSP. It is the VTScada analog of the CryptoAPI CryptAcquireContext call.
GetKeyParam	The CryptGetKeyParam function retrieves data that governs

	the operations of a key. It is the VTScada analog of the CryptoAPI's CryptGetKeyParam call.
Hash	Generates a hash – a text string of bytes – of the given string.
ImportKey	The ImportKey function transfers a cryptographic key from a key BLOB into a CSP (cryptography service provider). It is the VTScada analog of the CryptoAPI's ImportKey call.
SetKeyParam	The SetKeyParam function customizes various aspects of a session key's operations. The values set by this function are not persisted to memory and can only be used with in a single session. It is the VTScada analog of the CryptoAPI SetKeyParam call.

Database And Data Source

DBAdd	Executes in its own thread to add a record to a VTScada database and returns an indication of parameter errors.
DBGetStream	Executes in its own thread to convert a database to a stream, and returns an indication of parameter errors.
DBInsert	Identical to DBAdd, except will not update existing records.
DBListGet	Executes in its own thread to retrieve certain records from a list in a VTScada database and returns an indication of parameter errors.
DBListSize	Executes in its own thread to retrieve the size of a certain list in a VTScada database and returns an indication of parameter errors.
DBRemove	Executes in its own thread to remove a record from a VTScada database and returns an indication of parameter errors.
DBSystem	Creates a VTScada database and returns its value. The maximum field length is 65,523 characters. If the field length is longer than 65,523 characters, the DBSystem call will return invalid.

DBTrace	This is a trace engine that records live data to a SQL database.
DBTransaction	Executes in its own thread to perform a transaction on a VTScada database and returns an indication of parameter errors.
DBUpdate	Executes in its own thread to update a VTScada database from a given stream and returns an indication of parameter errors.
DBValue	Returns a certain value retrieved from a VTScada database.
ODBC	Performs an ODBC command and returns a (dynamically allocated) array if required.
ODBCBeginTrans	Indicates to a specified ODBC-compliant database that a transaction is to be started.
ODBCCommit	Indicates to a specified ODBC-compliant database that a transaction is to be committed.
ODBCConfigureData	Configures an ODBC data source and returns its error code.
ODBCConnect	Forms a connection to an ODBC-compliant database and returns the ODBC value associated with that database.
ODBCDisconnect	Stops a connection to the ODBC database.
ODBCRollback	Indicates to a specified ODBC-compliant database that a transaction is to be rolled back (discarded).
ODBCSources	Retrieves a list of ODBC data sources and returns it as a (dynamically allocated) array.
ODBCStatus	Returns the requested information about the last ODBC statement to execute.
ODBCTables	Retrieves a list of the tables present in an ODBC-compliant database and returns it as a dynamically allocated array.
RegisterCustomTable	Registers a name for a virtual database table and defines what information will be available from that table.

SQLQuery	A launched module that executes an SQL query on data in a VTS application.
TODBC	Performs an ODBC command; it is similar to ODBC except that it runs in its own thread.
TODBCBeginTrans	Indicates to an ODBC-compliant database that a transaction is to be started. TODBCBeginTrans is similar to ODBCBeginTrans, except that it runs in its own thread.
TODBCCommit	Indicates to an ODBC-compliant database that a transaction is to be committed. TODBCCommit is similar to ODBCCommit, except that it runs in its own thread.
TODBCConnect	Forms a connection to an ODBC database; it is similar to ODBCConnect except that it runs in its own thread (see "Comments" section for differences)
TODBCDisconnect	Stops a connection to the ODBC database; it is similar to ODBCDisconnect except that it runs in its own thread (see "Comments" section for differences).
TODBCRollback	Indicates to an ODBC-compliant database that a transaction is to be discarded. TODBCRollback is similar to ODBCRollback, except that it runs in its own thread.
TServerList	Executes in its own thread and creates a pointer to an array of all servers visible from this workstation; it returns a flag indicating its status upon completion.

DDE

DDE	Returns the value of the data for a specific item from a DDE server program. This function is a DDE client.
DDEPoke	Sends a value for a specific item to a DDE server program.
DDEShareAdd	Adds a new DDE share name to the SYSTEM.INI file or the registry and returns its own error code.
DDEShareDel	Deletes a DDE share name from the SYSTEM.ini file or the registry and returns its own error code.
SetDDEServer	Sets the DDE topic name for a window.

Dictionary

ClearVarMetaData	The opposite of SetVarMetaData, this statement removes all metadata associated with a variable.
Dictionary	Creates a database-like storage structure that provides efficient addition, retrieval and removal of information linked to key values.
DictionaryCopy	Create a new dictionary with contents identical to an existing dictionary. It is expectd that this function will be used rarely, since in most cases it will be more efficient to hand off a reference to a dictionary rather than build a duplicate of it.
DictionaryRemove	Removes a key / value pair from a dictionary, providing a means to regain memory space and remove data that is no longer needed.
GetKeyCount	Return a count of the number of keys stored by the given dictionary.
GetNextKey	Allows a linear search through a dictionary in place. i.e. without copying the contents to an array.
GetVarMetadata	Every variable object contains an embedded value. This function is used to retrieve those values.
HasMetaData	Tests whether a given variable is a dictionary. Since the default behavior of most operands and functions on dictionaries is to return just the value of the dictionary's root, this function provides the only means to determine whether or not a variable contains a dictionary.
IsDictionary	A synonym for HasMetadata. Tests whether the parameter is a dictionary.
ListKeys	Returns an array of all keys used within a dictionary. It is expectd that this function will be used primarily in the context of metadata (extended information attached to a variable). ListKeys also enables you to discover what is in a dictionary.
MetaData	If used with a variable which is not currently a dictionary, this

	command attached meta data to that variable, thereby creating a dictionary object. The primary purpose in this case is to provide a means of associating extended data with a variable.
RootValue	Retrieves the root value from a dictionary. This function will always attempt to return a value that is not itself a dictionary. If the value stored as the root of the given dictionary is also a dictionary, this function will return the root value from that second dictionary. Should all root values be other dictionaries (which would imply that the dictionary at the end of the chain must actually be an earlier dictionary) then RootValue will traverse the chain until it finds a root value which is an earlier dictionary (i.e. the end of the chain before it loops back) and will return that root value. This is the only situation where the command will return a dictionary as the result.
SetVarMetadata	Every variable object contains an embedded value. This function is used to set those values.

DLL

DLL	Returns a value of a type specified by its parameter from a call to Microsoft Windows™ dynamic link library using the C calling convention.
LoadDLL	Loads a Microsoft Windows™ dynamic link library.

Editor

AddEditorText	Inserts a text string into a text editor.
CurrentLine	Returns the text string that is the current line in an editor.
Editor	Displays an editor on the screen.
ForceEvent	Forces the editor to perform an action based on the information provided.
GoToOffset	Forces an editor to move to a location in its text.
MakeEditor	Returns an editor value which is used by an editor

MoveEditor	Moves the Editor to the given line and column.
SetEditMode	Sets the graphics edit mode for a window.
ZEditField	Draws a layered text edit field in a window and returns a status value.

Email

SendMail	Sends a string to an email server using the Simple Mail Transport Protocol (SMTP)
ValidateEmailAddrs	This subroutine validates a string of email addresses, and returns TRUE if the email addresses in the string are syntactically valid, or FALSE if they are not.

File I/O

CheckFileExist	(System Library) This subroutine checks for the existence of the specified file.
CheckPathExist	(System Library) This subroutine checks for the existence of the specified path.
CopyDir	(System Library) This subroutine recursively copies a directory's files and sub-directories down through the entire directory tree.
Dir	Performs a search in the given directory and returns an array of matching file names.
FileDialogBox	Displays a threaded system common file dialog box.
FileFind	Performs a recursive search down through the directory tree structure and returns an array of matching file names.
FileRootModule	Parses the document file that contains the given module to find the root module in that file. Returns the module value of the root module.
FileSize	Returns the size of a disk file in bytes.
FileStream	Returns a stream attached to a disk file or printer, and is suitable for use in SWrite.
FRead	Reads values from a formatted file and returns the number

	not read.
FWrite	Writes ASCII or binary data to a file and may also be used to create or delete a file. It returns the number of data items not written.
Get	Reads an array of historical data from a file (written by Save or SaveHistory) and returns the relative file position of the file entry following the last one read, or an error code.
GetFileAttribs	Returns information about the specified file.
GetHistory	Get History from a File written by Save or SaveHistory. This threaded function retrieves an array of data from a .DAT file for a certain time span. If the parameters to GetHistory are valid and an attempt is made to get the data, the return value is 0, otherwise, if no attempt is made to get the data, the return value is 1.
GetLog	This launched module returns an array of logged data. Marked for removal, but still in use as of VTS 10. Use GetTagHistory instead for all new code.
GetLogInfo	Interrogates a historical data file, or a set of historical data files, and returns overall time, date, and record count information either for the entire file(set), or for a specified time range.
LoadMIB	Loads a specified MIB or set of MIBs and returns a dictionary describing the hierarchy of the MIBs.
MkDir	Creates a directory on a disk and returns its own error code
ModifyConfiguration	Provides a safe way to write to configuration files.
ModuleFileName	Returns the full path (including the drive letter) and file name of the document (.SRC) file of a module.
ReadConfiguration	This function provides a safe way to read configuration files.
ReadINI	This subroutine read a variable entry from a Settings file or

	a buffer containing one and returns its value.
ReadSectINI	This subroutine read an entire section entry from a Settings file or a buffer containing one and returns a 2-dimensional array containing variable names and their values.
ReadX	Reads numeric data from a text file into the elements of an array.
ReadXY	Reads data points from a file into the elements of two arrays.
Rename	Renames an existing file.
ResyncDoc	Synchronizes the time and date for the document and .RUN files.
RmDir	Destroys a directory on a disk and returns its own error code.
RUNFileName	Returns the name of the .RUN file for a module including the full drive and path.
RUNFileVersion	Returns the minimum version of VTScada that can read the .RUN files produced by the current version.
Save	This threaded function stores data in a circular historical data file at times indicated by a condition and returns the record number last written to disk.
SaveHistory	This threaded function saves an array of data to a .DAT file for a certain time span.
SetCodeText	Will modify a source code file to replace the text for a given CodeValue with the new text.
SetFileAttribs	Sets the attributes of the specified file.
SpeakToFile	Executes on the speech thread to speak the supplied text to a .wav format audio file.
SplitPath	Breaks up a file path name into its components.
TempFileStream	Uses the OS tmpfile() function to create a temporary file on disk and to connect a stream to the temporary file. The temporary file is removed when the stream is closed or no

	longer referenced or if the VTScada process is terminated.
TGet	This threaded function reads an array of historical data from a file (written by Save or SaveHistory) and returns an indication of parameter errors.
WriteHistory	(Historian Manager Library) This subroutine writes a variable's value to a Settings file or a buffer containing one and returns its error code.
WriteSectINI	(System Library) This subroutine writes an entire section to a Settings file or a buffer containing one and returns its error code.
XMLWrite	Converts the instance of a type, as specified by XMLNodeTreeIn, into XML.

Error Manager

ReportError	Post error information to VTS Trace and optionally, to the display.

Graphics

4BtnDialog	(System Library) Draws a message dialog with up to 4 buttons and 3 lines of text and returns the number of the button that was pressed.
AlignSelected	Aligns selected graphic objects.
Bevel	(System Library) Draws a titled beveled box.
BitmapInfo	Returns information about an image.
Brush	Returns a brush value.
CaptureImage	Creates an image handle from a GUIStretch operation
CheckBox	(System Library) Draws a check box with (optional) label.
Click	Returns an indication of whether or not the mouse pointer is within a specified screen area and a particular button combination is being pressed.
ColorSelect	(System Library) Color Selection Tool. This module draws a

	color selection button and its accompanying display area.
Coordinates	Sets the VTScada screen coordinate limits (also called "world coordinates") used by the graphics functions.
CoordToPixel	Takes a specified coordinate pair within a given window and returns the overall, onscreen pixel location.
CopyObjects	Copies the code for selected drawing objects and returns it in a buffer.
Crop	Modifies an existing image, producing a new one that displays a sub-section of the original.
DateSelector	Displays a calendar, from which operators can select a date.
DialogInitPos	(System Library) Attempts to position a dialog so that it is not started beyond the left, right, top, or bottom of the visible screen.
DragHandle	Drags a selected graphic object's selected handle.
DrawArcPath	Draws an arc in any window.
DrawChordPath	Draws a chord in any window.
DrawEllipticalPath	Draw an ellipse in any window.
DrawPath	Draws a polygon in any window.
DrawPiePath	Draws a pie in any window.
DrawScale	(Meter Parts Library) Will draw a scale (i.e. tick marks) for a linear or radial type meter. These marks are images (normally lines) indicating the major and minor divisions of the entire scale. This function must be called inside a GUITransform in order to work properly.
Droplist	(System Library) Draws a droplist with (optional) title or bevel or both.
Edit	(System Library) Draws an edit field with (optional) title or bevel or both.
EditINI	(VTS Library) Draws an edit field from which a value of an application property in Settings.Startup may be set.

EditINICheckBox	(VTS Library) Draws a check box with which an application property in Settings.Startup may be set true or false.
Editor	Displays an editor on the screen.
FileDialogBox	Displays a threaded system common file dialog box.
FocusID	Returns the focus ID of the object in a window that currently has the input focus.
Folder	(System Library) Draws a tabbed folder dialog.
Font	Returns a font value.
FontDialog	Displays a threaded system common font dialog box.
Freeze	Freezes all or selected animated graphics in a window.
GetColorInfo	Returns the brush and pen information for a given graphic statement.
GetModuleRefBox	Get Module Reference Box
GetPathBound	Returns the bounding box coordinates for a path.
GetSelected	Returns a selected graphic item in a window.
GetSelectedInfo	Returns information about selected graphic item(s) in a window.
GetShapePath	Returns the path value which defines the shape of a polygon.
GetSystemColor	Returns the colors for the user-configured Windows↗ colors.
GetTrajectoryPath	Returns the Path value which defines the trajectory of a graphic object.
GetTransform	Returns the transform value applied to a graphic statement.
GetXformRefBox	Get Transform Reference Box. This function returns the reference box for any transform of a module.
Grid	Places a (lined) grid pattern on the screen.
GridList	(System Library) Draws a list in the style of a spreadsheet.
GUIArc	Draws an arc in a window and returns an indication when

	selected by a mouse button or the <ENTER> key.
GUIBitmap	Draws an image of any of the following formats in a window and returns an indication when selected by a mouse button or the <ENTER> key. Available formats include: BMP, EMF, WMF, APM, CUT, PCX, JPG, PNG, and TIF
GUIButton	Draws a push-button in a window and returns an indication when selected by a mouse button or the <ENTER> key.
GUIChord	Draws a chord in a window and returns an indication when selected by a mouse button or the <ENTER> key.
GUIEllipse	Draws an ellipse in a window and returns an indication when selected by a mouse button or the <ENTER> key.
GUIPie	Draws a pie-shaped wedge in a window and returns an indication when selected by a mouse button or the <ENTER> key.
GUIPipe	Deprecated. Use PathDraw. Draws a 3 dimensional, shaded pipe in a window and returns an indication when selected by a mouse button or the <ENTER> key.
GUIPolygon	Draws a multi-sided polygon in a window and returns an indication when selected by a mouse button or the <ENTER> key.
GUIRectangle	Draws a rectangle in a window and returns an indication when selected by a mouse button or the <ENTER> key.
GUIText	Draws formatted text in a window and returns an indication when selected by a mouse button or the <ENTER> key.
GUITransform	Applies a graphical transformation to all graphics in a module and returns an indication when selected by a mouse button or the <ENTER> key.
HScrollbar	(System Library) Draws a horizontal scrollbar and returns its position.
IconMarker	Creates the question mark and exclamation mark graphics, used to indicate questionable and manual data in a widget.

ImageArray	Reads an existing image handle and returns another one containing an image created from that handle by tiling the image a given number of times.
ImageSweep	Reads an existing image handle and returns another one containing an image created from that handle by tiling the image a given number of times along an arc-shaped path.
IPAddressList	Displays a list of IP address which can be added to or removed from.
LastSelected	Returns the most recently selected graphics statement.
LinearIndicator	(Meter Parts Library) Will draw a linear type indicator. A linear indicator can be drawn in 3 different ways: Scaled from min to current position, cropped from min to current position or as a line that moves to the current position. This function must be called inside a GUITransform in order to work properly.
LinearLegend	(Meter Parts Library) Draws a legend (i.e. the text labels) for a linear type meter. They are drawn in a line, either horizontally or vertically, with consistent spacing. This function must be called inside a GUITransform in order to work properly.
Listbox	(System Library) Draws a listbox with scrollbar (if required) and indicates the selected item.
MakeBitmap	Loads an image file of types BMP, EMF, WMF, APM, CUT, PCX, JPG, PNG, or TIF into memory and returns a handle to the result. Returns Invalid upon failure.
MapDraw	Draws a "slippy" map, showing a list of site tags as pins on that map.
ModifyBitmap	Reads an existing image handle and produces a new one with modifications. The original image is not altered. Returns invalid upon failure.
MoveWindow	Will move a window to the specified coordinates.
NextFocusID	Moves the focus position to a specific ID number.

Normalize	Returns a normalized value.
NumSelected	Returns the number of selected graphics statements in a window.
Output	Places formatted numbers or text on the screen.
Palette	Allows the color assignment for a color number to be changed, provided that Windows™ is running in 256 Color Mode.
PalStatus	Returns the current palette settings.
PAreaSelect	(VTS Library) Draws a droplist of area options with (optional) title and bevel.
ParentWindow	Returns the object value of the nearest non-child window.
PasteObjects	Pastes the code for multiple GUITransforms into a page source file.
Path	Returns a graphics path value.
PathDraw	Used within a Idea Studio handler to inform the engine that a path is being drawn, and to set the appearance of that path
PCheckBox	(VTS Library) Parameter Setting check box. This module draws a check box with (optional) label.
PColorSelect	(VTS Library) Draws a button that opens a color selection dialog and an area that displays the currently selected color.
PContributor	(VTS Library) Draws a split list displaying all contributors to a specific tag.
PDroplist	(VTS Library) Parameter Setting Droplist. This module draws a droplist with (optional) title and bevel.
PEditfield	(VTS Library) Parameter Setting Editfield. This module draws an editfield with (optional) title and bevel.
PEditName	(VTS Library) Tag name setting editfield. This module draws an editfield that is to be connected to the name of a tag. Should be used by all tag ConfigFolder modules for setting the name parameter.

Pen	Returns a pen value.
Pending	Returns the number of statements of a certain type pending.
Pick	Returns an indication of whether the locator (e.g. mouse) has had a specified change in its button status.
PIPAddressList	Uses an IPAddressList to set a parameter with a semicolon-delimited IP address list.
Pipe	Draw a Double Line
PIPListenerGroup	Draws a droplist of all available IP Listener Groups
PixelColor	Returns the color of a pixel in the window.
Plot	Displays a plot of a subsection of a numeric array in a particular area of the window.
PlotBuff	Displays a plot of a subsection of a buffer in a particular area of the window after converting the buffer to element values.
PlotXY	Displays a plot of a curve in the window given the X and Y values in two arrays.
Point	Returns a two-dimensional point, or location, in a window.
Popup	(Alarm Manager module) Causes an alarm pop-up dialog to be displayed.
PPageSelect	(VTS Library) Draws a titled, beveled droplist of pages in the system.
PRadioButtons	(VTS Library) Parameter Setting Radio Buttons. This module draws a set of labeled radio buttons with (optional) title and border.
PrintDialogBox	Displays a threaded system common printer selection dialog box.
ProgressBar	Displays a horizontal progress bar.
PSecBit	(VTS Library) Parameter Setting Security Bit. This module draws a titled, beveled droplist of options for setting the security bit.

PSelectObject	(VTS Library) Parameter Setting Select Tag Object Tool. This module draws a beveled, titled droplist of existing tags of a certain type and a new tag creation button.
PSpinbox	(VTS Library) Parameter Setting Spinbox. This module draws a spinbox with (optional) label.
PTypeToggle	(VTS Library) Parameter Setting Type Toggled Field. This module draws a beveled droplist or editfield with title that sets a tag or numeric value.
RadialIndicator	(Meter Parts Library) Will draw a radial type indicator that sweeps from a minimum angle to a maximum angle in the same fashion that a real radial meter would.
RadialLegend	(Meter Parts Library) Draws a legend (i.e. the text labels) for a radial type meter. They are drawn at a constant radius from the center point of the drawing coordinates, beginning at a defined minimum angle and ending at a defined maximum angle.
RadioButtons	(System Library) Draws a set of labeled radio buttons with (optional) title and border.
RootTransform	Returns the object value that contains the root transform applied to the given module.
Rotate	Returns a Rotate value, which specifies a rotation about a point.
SaveImage	Takes an image handle and saves it to an image file on disk.
SectionControl	(System Library) Creates a control that displays a variable number of sections. Visually, a section consists of a header and content. The control manages the layout and geometry for the sections and runs a caller-supplied module to display the section content.
SelectArea	Selects active graphics statements within a rectangular area in a window.
SelectCodePointer	Given a window object and a code pointer for an active graphics object within that window, this function adds the

	graphics object to the window's selection set.
SelectDAG	(i.e. Select Function) This function selects an active graphics DAG.
SelectGraphic	Selects an active graphics statement at a location in a window.
SelectHandle	Returns a pointer to a handle of selected graphics statements at a location in a window.
SelectHandleNum	Selects the given handle of selected graphics statements in a window.
SelectPath	Selects a path given its code pointer value.
SetCursor	Sets the mouse cursor type for the window.
SetHandle	Sets the position of graphics handles in a window.
SetModuleRefBox	Sets the reference box for a single instance of a module.
SetModuleRefBox	Sets the default reference box for a module.
SetRefRect	Sets the first four constant parameters of a layered graphic statement.
SetXLoc	Sets the X screen location of the locator (mouse).
SetYLoc	Sets the Y screen location of the locator (mouse).
ShowPage	When called with a Page Name, will cause that page to be displayed in the caller's display session context
SimulateMouse	Sets the pointer location and sends a button press with modifiers such as Ctrl, Shift or Alt. (mouse)
Spinbox	(System Library) Draws a spinbox with (optional) label.
SplitList	(System Library) Draws a split list (listbox with two columns) with a scrollbar if required and indicates the selected item.
SplitListSelector	(System Library) Draws a split view comprising two GridLists separated by control buttons. Items listed on the left may be selected to be transferred to the right. Items on the right may be returned to the left so long as they were originally listed on the left. Scrollbars will be drawn if required.

SplitTagSelector	Draws a split view comprising two GridLists separated by control buttons. Tag names listed on the left may be selected to be transferred to the right. Each GridList will have two columns: the name and the description of each tag in the list. Scrollbars will be drawn if required.
Tag	Returns a Tag value, which works like (and in place of) a Normalize value.
TagIconMarker	Draws an "IconMarker" in the center of its transform area, and optionally shows a blank icon when in editing mode.
TextAttribs	Returns graphic-related information about a text, given a font.
TextBox	(System Library) Displays a text string, breaking it into multiple lines at space or CRLF.
ToolBar	(System Library) Draws and maintains a toolbar and its buttons.
Trajectory	Move a Layered Graphic and return a Trajectory value.
UnselectGraphics	Will deselect all of the graphics in the specified window.
UnselectObject	Will deselect a statement in the context supplied.
UnTransform	Will undo a previous transform so that the module instance and everything it has called will not be transformed.
UpdateCoordinates	Will update a graphic statement's coordinates to the document file in which it is specified.
Vertex	Returns a Vertex value, which is a collection of 3 points and a mode.
VScrollbar	(System Library) Draws a vertical scrollbar and returns its position.
WinButton	Windows native button.
WinComboCtrl	Windows native "combo" control. A "combo" control is an enhanced form of drop list. Displays a child window containing a Windows combo control.
WinEditCtrl	Windows native edit control. This function returns a value

	indicating the status of an edit field.
WinTooltipCtrl	Windows native "tooltip" control. A "tooltip" is a pop-up text window that provides operational hints to users when the mouse pointer is rested over a tool or object.
WinYLoc	Returns the Y coordinate of the locator (mouse) in a window.
XLoc	Returns the X window coordinate of the locator (mouse).
YLoc	Returns the Y window coordinate of the locator (mouse).
ZBar	Draws a layered bar in a window.
ZBox	Draws a layered box in a window.
ZButton	Draws a layered button in a window and returns true when selected.
ZColorChange	Changes one color within a region to another color.
ZEditField	Draws a layered text edit field in a window and returns a status value.
ZGrid	Draws a layered point grid in a window.
ZLine	Draws a line between given x and y coordinates in a window.
ZPipe	Draws a layered three-dimensional pipe in a window.
ZText	Draws a layered text label in a window.

Help

EnableHelp	Enables you to enable or disable help file activation when the F1 key is pressed.
Help	Calls Windows™ help.
SetHelp	Sets the help file name and (optionally) the context identifier for the window containing the specified object.

Keyboard

FocusID	Returns the focus ID of the object in a window that currently has the input focus.

KeyCount	Returns the number of keys pressed since the state became active, either edited or non-edited.
KeyFake	Places a string of characters in a window's keyboard buffer.
Keys	Returns the most recently pressed keys from either the edited or non-edited keystrokes.
MatchKeys	Returns true if the specified keyboard keys have been pressed in the sequence given.
NextFocusID	Moves the focus position to a specific ID number.
WinMatchKeys	Returns true if the specified keyboard keys have been pressed in the sequence given in another window.
WinShiftKeys	Returns a value which contains the current status of the Shift, Control and Alt control keys.

Locator

Click	Returns an indication of whether or not the mouse pointer is within a specified screen area and a particular button combination is being pressed.
CurrentWindow	Returns the application window over which the mouse cursor currently rests.
LocCapture	Capture Locator Input. This statement captures all subsequent input from the locator device and routes it to a specific window.
LocSwitch	Returns the current status of the locator (mouse) buttons over the window which contains the LocSwitch statement.
Pick	Returns an indication of whether the locator (e.g. mouse) has had a specified change in its button status.
SetCursor	Sets the mouse cursor type for the window.
SetXLoc	Sets the X screen location of the locator (mouse).
SetYLoc	Sets the Y screen location of the locator (mouse).
Target	Returns an indication of whether the locator (e.g. mouse) is within a specified screen area.
WinLocSwitch	Returns the current status of the locator (mouse) buttons in a

	certain window and its ancestors.
WinTooltipCtrl	Windows native "tooltip" control. A "tooltip" is a pop-up text window that provides operational hints to users when the mouse pointer is rested over a tool or object.
WinYLoc	Returns the Y coordinate of the locator (mouse) in a window.
XLoc	Returns the X window coordinate of the locator (mouse).
YLoc	Returns the Y window coordinate of the locator (mouse).

Log

Get	Reads an array of historical data from a file (written by Save or SaveHistory) and returns the relative file position of the file entry following the last one read, or an error code.
GetHistory	Get History from a File written by Save or SaveHistory. This threaded function retrieves an array of data from a .DAT file for a certain time span. If the parameters to GetHistory are valid and an attempt is made to get the data, the return value is 0, otherwise, if no attempt is made to get the data, the return value is 1.
GetLog	This launched module returns an array of logged data. Marked for removal, but still in use as of VTS 10. Use GetTagHistory instead for all new code.
GetLogInfo	Interrogates a historical data file, or a set of historical data files, and returns overall time, date, and record count information either for the entire file(set), or for a specified time range.
GetTagHistory	(Historian Manager Library) Launched module that retrieves historical data for a tag. Replaces GetLog.
HistorianConnect	Opens a logging connection and controls the lifetime of all resources associated with that connection.
LogNTEvent	Logs events to the system event log.
SaveHistory	This threaded function saves an array of data to a .DAT file for a certain time span.

TGet	This threaded function reads an array of historical data from a file (written by Save or SaveHistory) and returns an indication of parameter errors.

Logic Control

Boolean	(System Library) Takes a valid Boolean test and returns the numeric equivalent.
Case	Selects one of a set of parameters for execution and returns its return value.
Cond	Returns one of two values depending upon the result of a conditional expression.
DoLoop	Executes a do-while loop in a script.
Execute	Executes a group of statements as a single entity in structures that would otherwise allow only one statement to be executed.
FALSE	Evaluates to 0 or tests whether an optional parameter evaluates to 0.
ForceState	Sets the next state to start when the action script completes.
IF	Performs an action; it changes the active state in a module instance or executes a script or does both.
IfElse	Executes one of its two parameters based upon a condition parameter.
IfOne	Check for an If One Condition. This function checks for a race condition in an action script and returns the value of the location.
IfThen	Conditionally Execute Statements. This statement executes a statement if the condition parameter is true.
Invalid	Return Invalid Value. This function always returns an invalid value.
PickValid	Attempts to return a valid value given a list of parameters.
TRUE	Evaluates to 1 or tests whether an optional parameter evaluates to 1.
WhileLoop	Repeatedly executes a parameter while a condition is true.

Math - Generic

ABS	Returns the absolute value of a numeric expression.

AMax	Array maximum. This function returns the maximum value in a sub-range of a numeric array.
AMin	Array minimum. This function returns the minimum value in a sub-range of a numeric array.
CommaFormat	Returns the text version of a number, formatted to use embedded commas.
Deriv	Returns the derivative (rate of change) of a value.
Edge	Test for a rising or falling edge.
Exp	Returns the natural antilogarithm of a numeric expression.
FFT	Performs a fast Fourier transform between time and frequency domains.
FitOffset	Linear regression offset. This function returns the offset or Y intercept of the least square curve fit of data in a pair of arrays.
FitR2	Returns the coefficient of determination (i.e. r2) for a linear curve fit. This number gives a measure of how correct the curve fit is.
FitSlope	Linear regression slope. This function returns the slope of the least square curve fit of data in a pair of arrays.
FormatInteger	Returns a text string which is a decimal value converted to one of binary, octal or hexadecimal formats.
Intgr	Time Integral. This function returns the time integral of a value.
Ln	Returns the natural logarithm (base e) of a value.
Log	Returns the common logarithm (base 10) of a number.
LValue	Left-hand Side Value. This function returns an indication of whether its argument can be used on the left-hand side of an assignment.
Max	Returns the maximum of a group of parameters.
Mean	Returns the mean (average) of a portion of a numerical array.
Min	Returns the minimum of a group of parameters.

Pow	Returns a number raised to a power.
Rand	Returns a random number between 0 and 1.
Scale	Returns a value that has been converted from one scale to another.
SDev	Returns the statistical sample standard deviation for a sub-section of an array.
Sqrt	Returns the square root of a number.
Sum	Returns the arithmetic sum of all the valid array elements in a specified portion of a numeric array.
SumBuff	Returns the summation of bytes in a buffer.
Tag	Returns a Tag value, which works like (and in place of) a Normalize value.
Variance	Returns the statistical sample variance for a subsection of an array.

Math - Rounding

Ceil	Returns the smallest integer greater than or equal to a number (the ceiling).
DeadBand	Returns the previous value of the first parameter until it changes by an amount specified by the second parameter.
Int	Integer Portion of Number. This function returns the portion of a number before the decimal point.
FormatNumber	Returns a compactly formatted version of a number, containing at least the specified number of significant digits.
Limit	Set Value Minimum and Maximum. This function returns a value that is limited both on the high and low ranges.
Step	Transforms a continuous value into discrete steps.

Math - Trigonometric

ACos	Calculates the trigonometric arc cosine in radians.
ASin	Returns the trigonometric arc sine in radians.

ATan	Returns the trigonometric arc tangent in radians.
Cos	Returns the trigonometric cosine of an angle in radians.
Sin	Returns the trigonometric sine of an angle in radians.
Tan	Returns the trigonometric tangent of an angle in radians.

Memory I/O

AddRead	Add a read request. This module is called by a tag to add a request to read a specific range of memory and set the resulting read data into the variable pointed to by the third parameter. The pointer may be a simple variable for one element read, or an array if more than one element is requested.
CopyIn	Copies data from an absolute RAM address and returns a buffer.
CopyOut	Copies data from a buffer to an absolute RAM address.
DelRead	Is called by a tag to delete an existing read request, as created by an AddRead.
In	Read I/O Byte. This function returns the byte read from an I/O port.
InWord	Read I/O Word. This function reads a 16 bit unsigned word from an I/O port.
MakeFixedBuff	Creates a buffer value which has its data stored at a specific memory address. Not supported under 64-bit VTScada.
MemIn	Returns a byte, word, or longword of RAM memory.
Memory	Returns the amount of memory that VTScada has acquired from the OS heap for internal use.
MemOut	Writes a byte, word, or longword of RAM memory.
MemTrace	Writes memory allocation information to a file.
New	Allocates memory for an array from RAM and returns a pointer to that array.
Out	Writes an 8 bit byte to an I/O port.
OutWord	Writes a 16 bit word to an I/O port.

Modem

CallerID	Takes a specified modem stream and returns the caller ID from the telephone system.
ModemCount	Returns the number of data modems configured and operational in the system.
ModemDev	Obtains the identifier for a modem sub-device. (In order to use the audio capabilities of a voice modem, the device identifier for the wave output device is required.)
ModemDial	Dials and attempts to connect to a remote modem and returns the modem stream or an error code.
ModemDigits	Controls the receipt of DTMF digits entered from the keypad of a telephone engaged on a voice call via a voice modem.
ModemList	Returns a list of the modems in the system. This function should be used to determine the correct parameter for the ModemStream or ModemDial functions.
ModemMedia	Enables you to determine the media mode of a serial stream open on a modem, and change it if necessary (for example, if you require the ability to be able to handle both incoming voice mode and data mode calls).
ModemStream	Open a serial stream on a modem and returns its status (prior to the connection being made), a modem stream (after the connection has been established), or an error code.
ModemTransfer	Transfers a modem call to another application and returns an indication of success.
PPPDial	Creates a PPP connection to a remote peer. The connection can be via a dial-up or direct device connection.
PPPHandles	Returns an array of all Point-to-Point Protocol handles on the local machine.
PPPDial	Describes the status of a PPP connection.

Module - Advanced

ActiveCode	ActiveCode returns the code value of the currently active statement in the given module instance.

AddModule	Adds a new module to an existing module and returns the value of the newly created module.
AddParameter	Adds an existing variable as a module parameter and returns the number of parameters in the module.
AdjustCode	Adjusts the offsets and sizes of items stored in the .RUN file within the document file.
CalledInstances	Returns the object values of module instances that are called by a particular module.
CanEditDoc	Returns an indication as to whether or not the document for the given module can be edited.
ChildDocs	Gets the module values for the root and all descendent modules that match the conditions defined by the second parameter.
CleanModule	Removes the flag that marks when a module that has been changed programmatically and would therefore have its changes saved to disk were this flag not cleared.
ClearModule	Deletes the contents (all variables and states) of a module without removing the module itself.
CodeText	Returns the text (or other information) for a given Code Value.
ConstCount	Returns the number of constant parameters in a function.
CoverageSnapshot	Captures the areas in a VTScada source file which have not executed along with summary statistic for each module in the file to extract code coverage information.
CreateModule	Creates a new module and returns a pointer to it.
CrossReference	Returns a linked list of structures representing all references to a specified variable or module.
DeleteModule	Deletes a module from the system.
FileRootModule	Parses the document file that contains the given module to find the root module in that file. Returns the module value of the root module.

FormalParms	Returns the number of formal parameters declared in a module.
GetModuleRefBox	Get Module Reference Box
GetModuleText	Returns information about a module's document file.
GetOneParmText	Returns the text for one parameter of a function.
GetOverrides	Returns an array of OpCodes and the module value that will run when each OpCode is executed
GetParameter	Returns the requested parameters as a constant, variable or code pointer.
GetParmText	Returns the text for all parameters of a function.
GetTransitText	Get Transition Document Text. This function returns information about the documentation of an action.
GetXformRefBox	Get Transform Reference Box. This function returns the reference box for any transform of a module.
GUITransform	Applies a graphical transformation to all graphics in a module and returns an indication when selected by a mouse button or the <ENTER> key.
ImportAPI	Imports objects of class API from a given module, for use in the calling module.
LoadModule	Loads a module from its .RUN files and returns a pointer to that module.
MCSInstance	Module Calling Structure Instance. This function returns the object value of a module called by another module.
MCSMod	Module Calling Structure Module. This function returns the module value from a line of code that calls that particular module.
ModuleFileName	Returns the full path (including the drive letter) and file name of the document (.SRC) file of a module.
NParm	Returns the number of parameters listed in a module instance.

NumParms	Returns the number of parameters of a statement.
OwningModule	Returns the module which contains a certain variable.
Pack	Packs a set of module parameters or an array of values into a stream, and returns the number of items that were not packed.
ParentModule	Returns the parent module of a module.
ParentObject	Returns the parent object of a module.
ParmToBuff	Returns a buffer of formatted numeric parameter values.
Priority	Sets the execution priority for a module, variable or object.
PType	Returns the actual type of parameter at an index.
RemoveParameter	Removes a parameter from a module's parameter list.
ResetParm	Can reset parameters that become latched.
ResyncDoc	Synchronizes the time and date for the document and .RUN files.
RootTransform	Returns the object value that contains the root transform applied to the given module.
RUNFileName	Returns the name of the .RUN file for a module including the full drive and path.
RUNFileVersion	Returns the minimum version of VTScada that can read the .RUN files produced by the current version.
SaveModule	Saves a module definition to its *.RUN file.
SectionControl	(System Library) Creates a control that displays a variable number of sections. Visually, a section consists of a header and content. The control manages the layout and geometry for the sections and runs a caller-supplied module to dis-play the section content.
SetCodeText	Will modify a source code file to replace the text for a given CodeValue with the new text.
SetModuleRefBox	Sets the reference box for a single instance of a module.

SetModuleRefBox	Sets the default reference box for a module.
SetModuleText	Sets the module's .SRC file information.
SetOneParmText	Sets the text for one parameter of a function.
SetOverride	Allows the overriding of OpCodes with a specified script module within a static module tree.
SetParameter	Sets a parameter in a statement.
SetParmText	Sets the text for the parameters of a function.
UnTransform	Will undo a previous transform so that the module instance and everything it has called will not be transformed.
UpdateCoordinates	Will update a graphic statement's coordinates to the document file in which it is specified.

Module - Basic

Call	Starts an instance of the module specified by its first parameter.
Caller	Takes a given object value for a module and returns the object value of the module by which it was called.
ChildInstances	Returns the object values of module instances that are children of a particular module instance (i.e. all objects whose parent is a specified object).
CriticalSection	Marks a section of a module as a critical section and will not allow interruption of its execution by other threads.
GetInstance	Returns the object value of a module instance.
GetTagList	Returns an array of tags, starting at a given point in the tag tree and including all child tags below that point, subject to the filtering parameters.
GetTagTypes	Returns an array of either the common names or the module names of all tag types.
GetReturnValue	Returns a module's return value.
HasReturnStatement	Examines a specified object to see if it is currently running a return statement in steady-state.

Instance	Limit Module Instances. This function limits the number of fixed module instances allowed to run simultaneously and returns the old limit
IsChild	Identify Child Module. This function returns an indication of whether one module is a child module of another.
Launch	Runs a module instance and returns a pointer to it.
LocalScope	Equivalent to Scope(Obj, Name, TRUE). LocalScope is a useful shortcut where the second parameter is not a constant string.
LocalVariable	LocalVariable(Name) is a shortcut for Scope(Self, Name, TRUE).
NumInstances	Returns the number of module instances currently running.
Parameter	Returns the value of (or may assign a value to) a parameter of a module, specified by the index.
PointList	Returns an array of tag names within the current scope, given the name of a tag type or group.
Return	Sets the return value for the module in which it is executed.
RootWindow	Returns the object value of the root (original) module displayed in the same window.
Scope	Performs a scope resolution and returns a reference to the requested member within a module or other object.
ScopeLocal	As above except that ScopeLocal will not look outside the given context.
Self	Returns the object value of the current module.
SetInstanceName	Set the name of an instance of a module.
SetReturnValue	Sets the return value of a specified object if the object is not currently running a Return() statement in steady state.
Slay	Stops a launched module, and possibly any parent modules.

SystemSelf	Returns the object value of the system module for the given application.
Thread	Launches a module in its own separate thread.
ThreadHistory	Returns in an array the history of execution for a specified thread.
ThreadIdle	Returns TRUE when the ToDo list for a given thread is empty.
ThreadList	Returns a two dimensional array containing the name and statement last executed by each VTScada thread.
ThreadName	Returns the name of a thread.
ThreadPriority	Allows advanced users to set a specified thread to one of six priorities, ranging from idle to time critical.
WCSubscribe	Working Copy Subscribe. After this function has been called, any configuration change will result in the specified callback subroutine being called.
Window	Opens a new window, starts a module inside, and eventually returns the module's value.

Network & Workstation

See also: RPC Manager Functions.

GetGUID	Creates a globally unique identifier or converts an existing GUID to another format.
GetHostByName	Calls the WinSock gethostbyname function and returns the host name, address(es) and alias names for the named computer.
GetPowerState	Returns a structure that describes the state of the power supply to the workstation. Relevant when the workstation is running on battery power.
LocalGroup	Returns an indication of whether the current Windows™ user is a member of the specified local group. LocalGroup interrogates only local groups, not domain groups.
OPCServer	Adds a new top-level branch to the VTScada OPC Server's

	namespace hierarchy.
Platform	Returns a twelve element structure that indicates the platform under which VTScada is currently running.
Redirect	Redirects a local device to network resource.
Register (RPC Manager)	(RPC Manager Library) This subroutine registers a service for RPC and returns a pointer to the variable containing the current RPC status of the service.
RunPack	(RPC Manager Library) Is unpacks and executes a set of RPCs from a stream constructed with PackRPC.
Send	(RPC Manager Library) This subroutine sends a message by invoking a remote procedure call (RPC).
ServerList	Returns a pointer to an array of all available servers visible from this workstation.
ServerSocket	Returns a server WinSock socket stream given a handle returned by a SocketServerStart function or an integer error code.
SetAllBlocks	(RPCManager Library) This subroutine executes on the client. It accepts all of the blocks and data for a service.
SetOPCData	Sets an item value in the VTScada OPC Server.
TCPIPReset	Shuts down and resets all TCP/IP functions.
TServerList	Executes in its own thread and creates a pointer to an array of all servers visible from this workstation; it returns a flag indicating its status upon completion.
WKSList	Generates a list of sub-paths from the query returned by the WKSPath function.
WKSPath	Given set of path components, generates a query path for use in the WKSStatus command.
WKSStatus	Sends a query to the Windows™ Performance Monitor interface (see image in WKSPath) and returns the result as a query handle.
WKStaInfo	Returns the characteristic information about this workstation.

ODBC

AddConnection	(ODBC Manager Library) Called to get the object value of an ODBC connection. If the connection does not exist, then a new ODBC DSN connection will be created and explicitly added to the ODBC Manager's internal list of DSN's
ErrMessage	(ODBC Manager Library) Returns a text message for the error code handed to it as a parameter
ODBC	Performs an ODBC command and returns a (dynamically allocated) array if required.
ODBCBeginTrans	Indicates to a specified ODBC-compliant database that a transaction is to be started.
ODBCCommit	Indicates to a specified ODBC-compliant database that a transaction is to be committed.
ODBCConfigureData	Configures an ODBC data source and returns its error code.
ODBCConnect	Forms a connection to an ODBC-compliant database and returns the ODBC value associated with that database.
ODBCDisconnect	Stops a connection to the ODBC database.
ODBCRollback	Indicates to a specified ODBC-compliant database that a transaction is to be rolled back (discarded).
ODBCSources	Retrieves a list of ODBC data sources and returns it as a (dynamically allocated) array.
ODBCStatus	Returns the requested information about the last ODBC statement to execute.
ODBCTables	Retrieves a list of the tables present in an ODBC-compliant database and returns it as a dynamically allocated array.
ResultFormat	(ODBC Manager Library) Subroutine to convert 2-d array as returned from query in the form, Arr[Field][Rec], to a normalized format of Arr[Rec][Field].
SQLQuery	A launched module that executes an SQL query on data in a VTS application.

TableSynch	(ODBC Manager Library) Synchronizes the fields matching a specified criteria within matching tables in two databases. Should be run as a called module, waiting for completion. Do not call as a subroutine.
TODBC	Performs an ODBC command; it is similar to ODBC except that it runs in its own thread.
TODBCBeginTrans	Indicates to an ODBC-compliant database that a transaction is to be started. TODBCBeginTrans is similar to ODBCBeginTrans, except that it runs in its own thread.
TODBCCommit	Indicates to an ODBC-compliant database that a transaction is to be committed. TODBCCommit is similar to ODBCCommit, except that it runs in its own thread.
TODBCConnect	Forms a connection to an ODBC database; it is similar to ODBCConnect except that it runs in its own thread.
TODBCDisconnect	Stops a connection to the ODBC database; it is similar to ODBCDisconnect except that it runs in its own thread.
TODBCRollback	Indicates to an ODBC-compliant database that a transaction is to be discarded. TODBCRollback is similar to ODBCRollback, except that it runs in its own thread.
Transaction	(ODBC Manager Library) Launches a transaction in the specified database connection. The transaction takes care of its own shut-down process.
TransactionCached	(ODBC Manager Library) Launches a transaction in the specified database connection. The transaction will be cached locally if it fails and then sent to the database after the next successful transaction. This module is designed to provide logging of values that must not be lost.

Printer

DefaultPrinter	Returns the Windows® default printer.
Print	Allows text to be printed.
PrintDialogBox	Displays a threaded system common printer selection dialog box.

PrintLine	Allows text to be printed and is followed by a carriage return-line feed to the printer.
PrtScrn	Prints the image in a window on the default Windows® printer and returns an error code.
Redirect	Redirects a local device to network resource.

Report

GetOutputTypes	Returns a list of available report output type plugins.
GetReportTypes	This subroutine returns a list of available report type plugins.

RPC Manager Functions

BinIP2Text	(RPC Manager Library) Returns a text representation of a specified binary IP in a printable format.
ConnectToMachine	(RPC Manager Library) This subroutine increments the usage count on the specified workstation and forces RPC Manager to attempt to establish a connection with the specified workstation if it is not already connected. Subroutine call only.
DisconnectFromMachine	(RPC Manager Library) This subroutine disconnects from a workstation by decrementing the usage count on the specified workstation and forcing the RPC Manager to attempt to establish a connection with the specified workstation if it is not already connected. Subroutine call only.
ForceServers	(RPC Manager Library) Sets the servership of an application service to a specific state.
GetClientDiverts	(RPC Manager Library) Returns a one-dimensional array of flags, indicating the divert status of each client.
GetClientGUIDs	(RPC Manager Library) Returns a one-dimensional array of the application GUIDs of the clients of the specified RPC service instance.
GetClientIPs	(RPC Manager Library) Returns a one-dimensional

	array of the IPs of the clients of the specified service instance.
GetClientList	(RPC Manager Library) Returns a one-dimensional array of the names of the clients of the specified service instance. Steady state or subroutine call.
GetClientMode	(RPC Manager Library) Returns a one-dimensional array of the modes of the clients of the specified service instance.
GetClientNodes	(RPC Manager Library) Returns a one-dimensional array of the object values of the MachineNodes of the clients of the specified service instance. Steady state or subroutine call.
GetInhibitedServiceList	(RPC Manager Library) Returns a one-dimensional array of the names of all services inhibited from RPCManager servership control.
GetInSyncServers	(RPC Manager Library) Returns a one-dimensional array of the names or IPs of the potential, synchronized servers for the given service.
GetIP	(RPC Manager Library) Returns an IP address for a workstation, given its name.
GetLocalIP	(RPC Manager Library) Returns an IP address for the local workstation that is known to the specified remote workstation.
GetLocalNumber	(RPC Manager Library) Returns the index of the local workstation down the prioritized server list for the named service. Steady state or subroutine call.
GetMachineNode	(RPC Manager Library) Returns the object value of the MachineNode for the specified name or IP. Steady state or subroutine call.
GetMakeAltPtr	(RPC Manager Library) Returns a pointer to a variable containing the Alternate status for the local service instance in the calling application for the specified ser-

	vice. Steady state or subroutine call.
GetRemoteVersion	(RPC Manager Library) Returns the version number of VTScada running on a specified workstation. Steady state or subroutine call.
GetServer	(RPC Manager Library) Returns the name of the currently active server for a specified service.
GetServerChanges	(RPC Manager Library) Launched by RPC Manager on a service server to obtain the service's synchronization data (i.e. called by RPC Manager during startup synchronization on a server to get the package of RPCs that create a synchronizable state on the client which is in step with the server).
GetServerMode	(RPC Manager Library) Returns the mode in which the current server for a specified service is running.
GetServerNumber	(RPC Manager Library) Returns the index down the prioritized server list of the current server for the specified service. Steady state or subroutine call.
GetServerSIDPtr	(RPC Manager Library) Returns a pointer to a variable that holds the session ID for the current server for the specified service.
GetServersListed	(RPC Manager Library) This subroutine returns a one-dimensional array of the names or IPs of the servers that has been derived from the "-Servers" section of the service configuration file.
GetServiceScope	(RPC Manager Library) Returns the service instance for a service.
GetSessionID	(RPC Manager Library) Returns the current session ID for a specified application on a workstation.
GetSocketStatus	(RPC Manager Library) Returns the connection status of either: 1) The machine node if the subnet is not valid, or 2) The socket that is on the specified subnet.
GetStatus	(RPC Manager Library) Returns a variable that holds

	the current service instance status for the specified service.
IsClient	(RPC Manager Library) Is Client of a Service. This subroutine returns an indication of whether or not a particular workstation is a client connected to a service. Returns 1 for the specified service if the specified machine is currently a client to the machine on which the IsClient() call is made.
IsMatch	(RPC Manager Library) Determines whether two names or IPs indicate the same workstation. This subroutine returns a "1" if the two names or IPs (any combination) refer to the same workstation.
IsPotentialServer	(RPC Manager Library) Is Potential Server for a Service. This subroutine returns an indication of whether or not the local workstation is a potential server for a service. Returns "1" if the local workstation can be a server for the specified service. IsPotentialServer should not be called in steady state.
IsPrimaryServer	(RPC Manager Library) Is Primary Server Active for a Service. This module returns an indication of whether or not the active server for a service is the primary server. Returns "1" if the local workstation is the current server for the specified service.
IsServiceReady	(RPC Manager Library) Is Primary Server Active for a Service. Only available in VTS 6. This module returns an indication of whether or not the specified server is in synchronization with the server instance. Returns "1" if the local instance is in synchronization with the server instance.
PackParms	(RPC Manager Library) This method packs supplied parameters into a stream. Subroutine call only.
PackRPC	(RPC Manager Library) Packs an RPC call and a set of parameters into a stream. Subroutine call only.

ReadLock	(RPC Manager Library) Attempts to acquire a Read lock for the specified service. Subroutine call only.
RecommendAlternate	(RPC Manager Library) Instructs RPC Manager that the local service instance does not consider itself a good server candidate.
RecommendPrimary	(RPC Manager Library) Instructs RPC Manager that the local service instance considers itself a good server candidate.
Register (RPC Manager)	Registers a service for RPC and returns a pointer to the variable containing the current RPC status of the service.
RunPack	(RPC Manager Library) Is unpacks and executes a set of RPCs from a stream constructed with PackRPC.
Send	(RPC Manager Library) This subroutine sends a message by invoking a remote procedure call.
SetDivert	(RPC Manager Library) Informs RPC Manager that the synchronization state of a service has been sampled during synchronization, and service RPCs for the specified client should be buffered until synchronization completes. Subroutine call only.
SetRemoteValue	(RPC Manager Library) This subroutine sets the specified variable within an application instance on a workstation to the specified value. Subroutine call only.
SetSyncComplete	(RPC Manager Library) Informs RPC Manager that service synchronization is complete as far as the local service instance is concerned. Subroutine call only.
TextIP2Bin	(RPC Manager Library) Returns the Binary representation of the specified IP.
UnpackData	(RPC Manager Library) This method unpacks a stream into an array or set of module instance parameters. Subroutine call only.

UnpackParms	(RPC Manager Library) This method unpacks a stream into the supplied parameters. Subroutine call only.
WriteLock	(RPC Manager Library) This subroutine attempts to require a Write lock for the specified service. Subroutine call only.

Security

Account Manipulation Methods

AddAccount	Creates a new account.
ModifyAccount	Modifies an existing account.
DeleteAccount	Removes an account.

Query Module

SecurityCheck	Examines the rules that apply to the current user or the named user to determine if the specified privilege has been granted.
BuildFullName	If a namespace and namespace delimiter are being used, returns the full, namespace-qualified name of the specified account.
GetFullName	Returns the full, namespace-qualified name of the caller's account.
GetGroupName	Returns the namespace of the caller's account.
GetUserName	Returns the user name of the caller's account.
GetAccountID	Returns the account ID of the named account.
GetAccountInfo	Returns one or more AccountData structures.
IsLoggedOn	Returns TRUE if the calling user is logged on, else FALSE.
IsSecured	Returns TRUE if the application has any user accounts defined, else FALSE.
IsSuspended	Returns TRUE if the user's account is suspended, else FALSE.

UIErrorToText	Returns a text string corresponding to the error code provided.

VTScada Authentication Module

AlternateIdCheck	Searches the accounts for an account whose AltID matches the parameter value.
AlternateLogon	Either creates, or attempts to log in using an alternate ID value.
AlternateLogoff	Synonym for LogOff().
Authenticate	Authenticates the NameSpace, UserName and Password.
QuietLogon	Authenticates the authorization token (AuthToken) and, if successful logs the calling user session on as the user specified in the AuthToken.
LogOff	Logs the calling user session off.
UserCredChange	The return value will increment each time there is a change in the user session's logged-in user or their password.

Windows Authentication Module

UserLogonDialog	Returns the string value of the LDAP default naming context for the host machine domain.
WindowsLogon	Authentication request to Windows Authentication services.

User Interface Module

UserLogonDialog	Launches the Logon dialog.

Serial Port

COMPort	Opens a serial port and handles all interrupts and asynchronous events for that serial port, including transmission, reception, and control. It returns its own error code. Please note that the SerialStream function is generally preferred in many situations; however, ComPort continues to be supported.

PPPDial	Creates a PPP connection to a remote peer. The connection can be via a dial-up or direct device connection.
SerBreak	Sends a break character to a serial port.
SerCheck	Check Serial Port. This function returns the immediate or cumulative serial port status.
SerialStream	Returns a serial stream that can be used in any of the serial port functions or with any of the stream functions. Please note that the ComPort function (which functions somewhat differently than the SerialPort function) may also be utilized.
SerIn	Get Serial Port Byte. This function returns the next byte in the receive buffer.
SerLen	Serial Port Buffer Length. This function returns the number of bytes currently in the receive or transmit buffers.
SerOut	Send Serial Port Byte. This statement sends a byte to the transmit buffer.
SerRcv	Serial Port Receive. This function returns a buffer containing a string read from the receive buffer.
SerRTS	Sets or clears the RTS line on a serial communication port.
SerSend	Serial Port Send. This function writes a string to the transmit buffer and returns the number of bytes written.
SerStrEsc	Serial Port Receive With Escape. This function reads the receive buffer until an escape code is encountered and returns the final offset in the buffer.
SerString	Serial Port String Receive. This function reads the receive buffer until a string is encountered and returns the final offset in the buffer.

Software And Hardware

CommandLine	Returns any command line arguments as a text string.
DriveInfo	Returns information about a disk drive.
GetConfiguration	Returns the configuration parameters from the license key for this copy of VTScada.

IsRunning	Check if a Program is running. This function returns an indication of whether a certain program is currently running on the same computer.
LoadDLL	Loads a Microsoft Windows™ dynamic link library.
LoadMIB	Loads a specified MIB or set of MIBs and returns a dictionary describing the hierarchy of the MIBs.
MACID	Enumerates and returns the MAC IDs registered on a particular machine.
Platform	Returns a twelve element structure that indicates the platform under which VTScada is currently running.
PriorityWeight	Sets the system-wide weighting for priority values.
ProcInfo	Returns basic information about the VTScada process.
Profile	Returns an array profiling the execution of statements in the application.
Redirect	Redirects a local device to network resource.
SerialNum	Returns the serial number of the copy of VTScada running.
Spawn	Runs another Windows™ program.
Stop	Causes the immediate termination of VTScada, closing all windows.
ThreadList	Returns a two dimensional array containing the name and statement last executed by each VTScada thread.
ThreadName	Returns the name of a thread.
Version	Returns the version number of the copy of VTScada currently running.
VersionRequired	Returns the version number of VTScada that is required to execute any .RUN files loaded since the last execution of this statement.
VStatus	Returns the video board and screen characteristics for VTScada.
WKSPath	Given set of path components, generates a query path for use

	in the WKSStatus command.
WKSStatus	Sends a query to the Windows™ Performance Monitor interface (see image in WKSPath) and returns the result as a query handle.
WKStaInfo	Returns the characteristic information about this workstation.

Speech And Sound

AudioFileLength	(System Library) Returns the length of a RIFF format Wave file in seconds.
Beep	Causes a tone to sound on the computer's internal speaker.
Configure	Is used to define how a speech stream will sound and where it will be heard.
GetDevices	Runs in the VoiceTalk thread and returns a list of devices available on a SAPI text-to-speech stream.
GetVoices	Runs in the VoiceTalk thread and returns a list of voices available on a SAPI text-to-speech stream.
MuteSound	(Alarm Manager module) This subroutine is used to turn off alarms sounds for all alarms, both current and future.
Play	Plays a multimedia sound file as installed in the operating system. It differs from Sound in that it is a steady-state statement and is supported by VTScada Internet Client.
Reset	Immediately stops a speech stream and cancels any buffered speech.
ShowLexicon	Displays a SAPI text-to-speech engine lexicon dialog to permit modification of pronunciation. This function will return immediately, and the lexicon window will be managed in its own thread, preventing the calling thread from being blocked.
Sound	Plays a multimedia sound file as installed in the operating system.
Speak	Executes on the speech thread to speak the supplied text through a specified SAPI text-to-speech stream.
SpeakToFile	Executes on the speech thread to speak the supplied text to a

	.wav format audio file.
VoiceTalk	Opens and returns a handle to a SAPI text-to-speech stream.

State

ActiveState	ActiveState returns the code value of the currently active state in the given module instance.
AddOptional	Adds a new statement to an action script and returns its own error code.
AddPageToApp (Obsolete)	Creates a new application page.
AddState	Adds a new state to an existing module and returns its state value.
AddStatement	Adds a new statement to an existing state and returns its own error code.
ClearState	Deletes all of the statements in a state.
DeleteOptional	Deletes a statement from an action script.
DeleteState	Deletes a state from a module.
DeleteStatement	Deletes a statement from a state.
FindAction	Returns an action from the list of actions in a state.
FirstState	Sets the first state in a module.
ForceState	Sets the next state to start when the action script completes.
GetState	Returns the code value for the specified state.
GetStatement	Returns the code value for the specified statement.
GetStatementNum	Returns the statement number for the specified statement.
GetStateText	Returns the text for the specified state.
ReplaceStatement	Replaces a statement with another statement.
SetStateText	Sets the information about the text of a state in a .SRC file.
SetTransfer	Sets the destination for an action.

StateList	Returns a list of states for a module.
StatementInstance	Takes a given code value and object and returns a code pointer value for that instance.
StateName	Returns the text name of the given state.
SubStatementIndex	Returns the index of a function within the statement where it is called.

Stream And Socket

BlockWrite	Writes a block of data to a stream.
BuffStream	Returns an in-memory read/write (expanding) buffer stream.
ClientSocket	Opens a client WinSock-compliant socket stream and returns a stream value, or a numeric error code.
CloseStream	Closes and flushes any type of open stream and returns its own error code.
CommaFormat	Returns the text version of a number, formatted to use embedded commas.
DBGetStream	Executes in its own thread to convert a database to a stream, and returns an indication of parameter errors.
FileStream	Returns a stream attached to a disk file or printer, and is suitable for use in SWrite.
Flush	Pushes the data in all software caches associated with a FileStream directly to the physical media.
GetStreamLength	Returns the present length of a stream in bytes.
GetStreamType	Returns a type indication for a stream.
GetToken	Reads the next token from a stream and returns the token type.

In	Read I/O Byte. This function returns the byte read from an I/O port.
Pack	Packs a set of module parameters or an array of values into a stream, and returns the number of items that were not packed.
PackParms	(RPC Manager Library) This method packs supplied parameters into a stream. Subroutine call only.
PackRPC	(RPC Manager Library) Packs an RPC call and a set of parameters into a stream. Subroutine call only.
PeekStream	Returns a string of bytes from a stream without removing them from the stream.
PipeStream	Returns a stream based on an operating system named pipe.
Read	(VTSDriver Library) Used by a tag to create a request for a single read of a given driver address. (This is in contrast to the polled read request of the AddRead function).
ReadBlock	(VTSDriver Library) Is launched to read a block of data from the PLC. It maintains a linked list of pointers to tag values with their absolute offset into the PLC file being read by this instance.
RunPack	(RPC Manager Library) Is unpacks and executes a set of RPCs from a stream constructed with PackRPC.
Seek	Changes and returns the current position within a stream. The return value is the current stream position after the seek is done.
SerialStream	Returns a serial stream that can be used in any of the serial port functions or with any of

	the stream functions. Please note that the ComPort function (which functions somewhat differently than the SerialPort function) may also be utilized.
ServerSocket	Returns a server WinSock socket stream given a handle returned by a SocketServerStart function or an integer error code.
ShiftStream	Inserts or deletes characters from a stream and returns its own error code.
SocketAttribs	Returns information about a TCP/IP socket's attributes.
SocketPingSetup	Starts the transmission of automatic keep-alive "ping" messages through a socket stream.
SocketServerEnd	Ends a TCP/IP socket server.
SocketServerStart	Starts a TCP/IP or UDP socket server and returns a handle to it.
\SocketServerManager\Register	Register a station with a group. (VTScada Programmer's Guide)
\SocketServerManager\UnRegister	Unregister a station from a group. The station must be unregistered whenever GroupName, or StationKey changes. (VTScada Programmer's Guide)
SocketWait	Wait for Socket Connect. This function returns true when a client connects to a socket offered by a socket server.
SRead	Reads values from a formatted stream and returns the number of values not read.
StreamEnd	Returns whether or not a stream is at the end.
SWrite	Performs a formatted write of ASCII or binary

	data to a pre-existing stream and returns the number of data items not written.
TempFileStream	Uses the OS tmpfile() function to create a temporary file on disk and to connect a stream to the temporary file. The temporary file is removed when the stream is closed or no longer referenced or if the VTScada process is terminated.
Unpack	Unpacks a set of values from a stream into a single dimensional array or a set of variables referenced by object parameters, and returns the number of items unpacked.
UnpackData	(RPC Manager Library) This method unpacks a stream into an array or set of module instance parameters. Subroutine call only.
UnpackParms	(RPC Manager Library) This method unpacks a stream into the supplied parameters. Subroutine call only.
Write	(VTSDriver Library) Used by a tag to create a write request to a driver address.

String And Buffer

ArrayToBuff	Returns a buffer containing the numeric data from an array.
BinIP2Text	(RPC Manager Library) Returns a text representation of a specified binary IP in a printable format.
BuffOrder	Reverses the order of groups of bytes in a buffer, and returns a new (rearranged) buffer.
BuffRead	Reads values from a formatted buffer and returns its own error code.
BuffStream	Returns an in-memory read/write (expanding) buffer stream.
BuffToArray	Reads an array from a formatted buffer containing numerical data and returns the number of elements read.

BuffToHex	Convert a buffer of numeric data to a hexadecimal value.
BuffToParm	Convert buffer of numeric data to parameters. This function reads module parameters from a formatted buffer containing numerical data and returns the number of data items read.
BuffToPointer	Converts a buffer of numeric data to array of pointers. This function reads from a formatted buffer containing numeric data, writes to locations specified by an array of pointers, and returns the number of elements read.
BuffWrite	Writes formatted values to a buffer and returns the number of values not written.
CharCount	Returns the number of bytes in a section of a buffer that matches a search character.
ClipboardGet	Returns the current contents of the system clipboard as a string. This function enables an application to perform text "paste" operations.
ClipboardPut	Set the current contents of the system clipboard to a string. This function enables an application to perform text "copy" or "cut" operations.
Concat	Returns the text value that is the concatenation of all the text parameters.
CRC	Returns the cyclic redundancy check (CRC) value for a buffer.
CRCTable	Returns a buffer containing a CRC table.
Deflate	Compresses/decompresses a buffer of data using the DEFLATE algorithm, and returns the compressed/decompressed data.
Diff	Compares two buffers and generates a third buffer containing formatted instructions describing how the first buffer can be modified so that it will match the second. This will perform a delimited difference unless the ChunkSize parameter is set to 1 or greater.
Format	Returns a text string corresponding to numbers in a specified format.

FormatInteger	Returns a text string which is a decimal value converted to one of binary, octal or hexadecimal formats.
FRead	Reads values from a formatted file and returns the number not read.
FWrite	Writes ASCII or binary data to a file and may also be used to create or delete a file. It returns the number of data items not written.
GetByte	Returns a single byte from a buffer.
HexToBuff	Converts a hex string to a binary buffer.
IPAddressList	Displays a list of IP address which can be added to or removed from.
KeyFake	Places a string of characters in a window's keyboard buffer.
Keys	Returns the most recently pressed keys from either the edited or non-edited keystrokes.
Locate	Locates a text string, returning the offset of the first matching string in a buffer.
MakeBuff	Creates a buffer and returns its address.
MakeFixedBuff	Creates a buffer value which has its data stored at a specific memory address. Not supported under 64-bit VTScada.
ParmToBuff	Returns a buffer of formatted numeric parameter values.
PatternMatch	Compares a string against a reference pattern and returns true if the string matches the pattern. Along with literal characters, PatternMatch currently supports the * and ? wildcard characters within the reference pattern.
PlotBuff	Displays a plot of a subsection of a buffer in a particular area of the window after converting the buffer to element values.
PointerToBuff	Returns a buffer containing the numeric data from the variables pointed at by each element of the array.
Replace	Performs a search and replace operation on a buffer and returns the resulting buffer.

Reverse	Returns its parameter with the byte order reversed.
SerLen	Serial Port Buffer Length. This function returns the number of bytes currently in the receive or transmit buffers.
SerOut	Send Serial Port Byte. This statement sends a byte to the transmit buffer.
SerString	Serial Port String Receive. This function reads the receive buffer until a string is encountered and returns the final offset in the buffer.
SetByte	Writes a single byte to a buffer.
StrCmp	Performs a case sensitive comparison of two text expressions and returns an indication of whether the first string is greater than, less than or equal to the second.
StrICmp	Case Insensitive Text Comparison
StrJustify	Reformats a string to have maximum line length.
StrLen	Returns the length of a text string.
SubStr	Returns a string that is a portion of another string.
SumBuff	Returns the summation of bytes in a buffer.
TextIP2Bin	(RPC Manager Library) Returns the Binary representation of the specified IP.
TextSearch	Returns the array index of the first occurrence of the given text key in an alphabetically ordered array.
TextSize	Returns the size in characters of the definition text of a desired item.
ToLower	Returns a text string with all the characters converted to lower case.
ToUpper	Returns a text string with all the characters converted to upper case..
Write	(VTSDriver Library) Used by a tag to create a write request to a driver address.

Time And Date

In addition to the functions listed here, a utility module, TimeUtils maintains an updated list of timestamps such as "start of last week in UTC" and "end of last month in local time". These are available by calling \TimeUtils\Periods or \TimeUtils\GetPeriods.
Note that, although the start and end timestamps are available in UTC, the time periods themselves are based on local time. For example, "PreviousWeek" means the previous week in local time. The stored values are updated each time a new time period starts.

Examples:

```
\TimeUtils\Periods["PreviousWeek"]["StartTimeUTC"]
```

called in steady state will contain the UTC start time of the previous week based on the StartOfWeek application property.
Stored time periods include PreviousDay, PreviousWeek and PreviousMonth.

```
\TimeUtils\GetPeriods("PreviousMonth", Timestamp)
```

Returns a dictionary containing the start and end times of the previous month relative to Timestamp, in local time and UTC.

AbsTime	Absolute time. This function returns true when a fixed time has been reached.
ConvertTimeStamp	Converts a timestamp from one time zone to another.
CurrentTime	Returns the number of seconds, in local or UTC time, since midnight of January 1, 1970 (where "midnight" is 00:00).
Date	Returns a text string giving the date that corresponds to the number of days since January 1, 1970.
DateNum	Returns the number of days since January 1, 1970 for a given date.
DateSelector	Displays a calendar, from which operators can select a date.
Day	Returns the day of the month for a given date number.
Month	Returns the month for a given date number.

Now	Returns the current time in seconds since midnight.
RTimeOut	Cumulative Timer. This function returns true when the total time that an expression is true reaches the specified value.
Seconds	Returns the number of seconds since midnight of the current day.
SetClock	Sets the VTScada system clock and calendar.
Time	Returns a formatted string for a time of day.
TimeArrived	Indicates whether a given time has occurred.
TimeOut	Returns true when the uninterrupted time that an expression is true reaches the specified value.
TimeZone	Returns information on the current time zone setting of the machine.
TimeZoneList	Returns an array of names of time zones that are available on the system.
Today	Returns the current number of days since January 1, 1970.
Year	Returns the year for a given date number.

Variable

Accumulate	Adds a tag and a value to be counted for that tag, to a named accumulator.
AddVariable	Adds a new variable to an existing module and returns its variable value.
Cast	Takes a value and returns a different type of value, if possible.
Change	Returns a true when the value of the first parameter changes by at least the value of the second parameter.
ChangePersistentSize	Changes the space allocated in the persistent value (.VAL) file for a variable.
CheckTagGroup	Returns TRUE or FALSE according to whether a tag is in the specified group.

DeleteVariable	Deletes a variable from a module.
Dictionary	Creates a database-like storage structure that provides efficient addition, retrieval and removal of information linked to key values.
DictionaryCopy	Create a new dictionary with contents identical to an existing dictionary. It is expectd that this function will be used rarely, since in most cases it will be more efficient to hand off a reference to a dictionary rather than build a duplicate of it.
Edge	Test for a rising or falling edge.
FindVariable	Searches for a variable by text name and returns a variable value.
GetDefaultValue	Returns a variable's default value.
GetReferencedValues	Collect all dynamically referenced values in the call tree rooted at the parameter and return them in an array.
GetSessionContainers	Returns an array of the names of tags that are "container" tags that exist at any level under the given context (parent) tag
GetSessionContainerTags	Returns a dictionary of tag items below a given context (parent) tag
GetUserID	Returns the name of the user for the current session.
GetValue	Returns the current count of the values within an accumulator dictionary.
GetVariableText	Returns information about the documentation of a variable.
GetVariableType	Returns the type, BASEVALUE, stored within a variable.
GetVarMetadata	Every variable object contains an embedded value. This function is used to retrieve those values.
HasMetaData	Tests whether a given variable is a dictionary. Since the default behavior of most operands and functions

	on dictionaries is to return just the value of the dictionary's root, this function provides the only means to determine whether or not a variable contains a dictionary.
Invalid	Return Invalid Value. This function always returns an invalid value.
IsEqual	Will return TRUE if the parameter values are equivalent, or if both are invalid.
IsDictionary	A synonym for HasMetadata. Tests whether the parameter is a dictionary.
Latch	Latch On or Off. This function allows a transient change of a variable to be captured. Its return value is determined by the rules listed in the comments section.
ListKeys	Returns an array of all keys used within a dictionary. It is expectd that this function will be used primarily in the context of metadata (extended information attached to a variable). ListKeys also enables you to discover what is in a dictionary.
ListVars	Returns a list of variables.
LValue	Left-hand Side Value. This function returns an indication of whether its argument can be used on the left-hand side of an assignment.
MakeNonPersistent	Takes a variable and makes it not persistent.
MakeNonShared	Takes a shared variable and makes it not shared.
MakePersistent	Takes a variable and makes it persistent (static).
MakeShared	Takes a variable and makes it shared.
MetaData	If used with a variable which is not currently a dictionary, this command attached meta data to that variable, thereby creating a dictionary object. The primary purpose in this case is to provide a means of associating extended data with a variable.

NumSets	Returns the number of statements that are currently active in setting a particular variable.
NumVariables	Returns the number of variables in a module.
Parameter	Returns the value of (or may assign a value to) a parameter of a module, specified by the index.
PersistentSize	Returns the size in bytes of a variable's persistent value size in the persistent value (.VAL) file.
PID	Perform PID Controller Function. This function returns a control value to maintain a parameter at a given setpoint.
Priority	Sets the execution priority for a module, variable or object.
PType	Returns the actual type of parameter at an index.
PTypeToggle	(VTS Library) Parameter Setting Type Toggled Field. This module draws a beveled droplist or editfield with title that sets a tag or numeric value.
RootValue	Retrieves the root value from a dictionary. This function will always attempt to return a value that is not itself a dictionary. If the value stored as the root of the given dictionary is also a dictionary, this function will return the root value from that second dictionary. Should all root values be other dictionaries (which would imply that the dictionary at the end of the chain must actually be an earlier dictionary) then RootValue will traverse the chain until it finds a root value which is an earlier dictionary (i.e. the end of the chain before it loops back) and will return that root value. This is the only situation where the command will return a dictionary as the result.
SetDefault	Sets the default value for a variable.
SetVariableClass	Sets the class number of a variable and returns its previous class number.

SetVariableText	Sets the information about the documentation of a variable in the .SRC file.
SetVariableType	Sets the data type for the variable, so that only values of that data type can be stored in the variable.
StaticSize	Returns the size of a given variable, provided that the variable is static.
Struct	Returns a dictionary, using the provided parameters as keys.
Tag	Returns a Tag value, which works like (and in place of) a Normalize value.
Toggle	Returns its previous status value except when its parameter changes from a false to a true, in which case it changes its value.
Valid	Returns true if the parameter is valid.
ValueType	Returns the type of value passed to it.
Variable	Accesses a variable by its text name; its return value is optional.
VariableClass	Returns the class of a variable.
Watch	Watches its parameters and returns true when any of their types or values change.
WatchArray	Watches an array and returns true if any of its elements' types or values change.
WatchForTagChanges	Watches for tag changes, either external or local.
WCSubscribe	Working Copy Subscribe. After this function has been called, any configuration change will result in the specified callback subroutine being called.

VTScada Internet Client

IsVICSession	Returns TRUE to indicate that a call is being made from a VTScada Internet Client session.
NotifyVIC	Sends a message to the VTScada Internet Client (VIC). The mes-

	sage sent depends on the parameter given to the function.
SetVicParms	Sets parameters for the VTScada Internet Client.
VICInfo	Provides information about the currently connected VTScada Internet Clients.
VICMessage	Transmits a message to one or all currently connected VTScada Internet Client sessions. The message is displayed in a dialog box on the VIC computer.

Window

ActiveWindow	ActiveWindow returns the object value of the root module instance in the current active window.
Click	Returns an indication of whether or not the mouse pointer is within a specified screen area and a particular button combination is being pressed.
Cls	Clears the screen and sets its background color.
Coordinates	Sets the VTScada screen coordinate limits (also called "world coordinates") used by the graphics functions.
CoordToPixel	Takes a specified coordinate pair within a given window and returns the overall, onscreen pixel location.
Crop	Modifies an existing image, producing a new one that displays a sub-section of the original.
CurrentWindow	Returns the application window over which the mouse cursor currently rests.
DelPageFromApp	Deletes a system page from an application.
FocusID	Returns the focus ID of the object in a window that currently has the input focus.
Freeze	Freezes all or selected animated graphics in a window.
GetXformRefBox	Get Transform Reference Box. This function returns the reference box for any transform of a module.
LocCapture	Capture Locator Input. This statement captures all subsequent input from the locator device and routes it to a spe-

	cific window.
MoveWindow	Will move a window to the specified coordinates.
ParentWindow	Returns the object value of the nearest non-child window.
RootWindow	Returns the object value of the root (original) module displayed in the same window.
SetCursor	Sets the mouse cursor type for the window.
SetEditMode	Sets the graphics edit mode for a window.
SizeWindow	Changes the visible size of a window on the screen.
UnselectGraphics	Will deselect all of the graphics in the specified window.
VStatus	Returns the video board and screen characteristics for VTScada.
WinButton	Windows native button.
WinComboCtrl	Windows native "combo" control. A "combo" control is an enhanced form of drop list. Displays a child window containing a Windows combo control.
Window	Opens a new window, starts a module inside, and eventually returns the module's value.
WindowClose	Returns true if an attempt to close the window is made.
WindowOptions	Alters the options on a window once it has been opened.
WindowSnapshot	Creates an image file containing a screen capture of the specified window.
WinLocSwitch	Returns the current status of the locator (mouse) buttons in a certain window and its ancestors.
WinYLoc	Returns the Y coordinate of the locator (mouse) in a window.
XLoc	Returns the X window coordinate of the locator (mouse).
YLoc	Returns the Y window coordinate of the locator (mouse).

XML

GetXMLNodeArray	Searches the result returned from XMLParse and returns an array of XMLNode values of a given type.

RemWSDL	Disconnects a Realm from a WSDL file and the associated set of VTScada modules, cleaning up any resourced used by that web service.
SetWSDL	Connects a Realm with a WSDL file and a set of VTScada modules in order to enable a web service interface.
XMLAddSchema	Adds a schema to an XML Processor.
XMLCloneNode	Clones an existing XMLNode, optionally adding additional members.
XMLCreateNode	Creates a new XMLNode.
XMLDeleteMember	Deletes a member from an XMLNode in-place.
XMLGetNode	Returns an XMLNode from a tree.
XMLParse	Parses the supplied XML using the specified XML Processor.
XMLProcessor	Creates a new XML Processor.
XMLWrite	Converts the instance of a type, as specified by XMLNodeTreeIn, into XML.

Usage Rules for Functions

VTScada code runs in two modes: Script or Steady State. Many functions will work in only one mode. The "Usage" line in each function description tells you the mode where the function can be used.

Note: Just because a function can be used in a given situation, doesn't mean that it should be. For example:
* It makes no sense to put a graphics function into a Calculation tag's expression.
* MatchKeys will capture keystrokes only when used in a window or page, not in a service or Calculation tag.
* Script-mode functions can be used for optimized tag parameter configuration, but many are not appropriate in that context.

If you are writing...

General Expressions (Calc. tags)
If you are writing an expression for a Calculation tag, or anywhere that you have the option "Constant / Expression / Tag":

If the function is marked as "Script Only" then you cannot use it here.
If the function works in Steady State, then it will compile when used in a Calc tag expression, but it may or may not be useful there. For example,

Tag Parameter Expressions - Optimized
Only functions that can be used in Script may be used for optimized tag parameter expressions. These expressions are evaluated as the tag is initialized, then not run again during normal operations. You cannot use Steady State-only functions in this situation.

Tag Parameter Expressions - Not Optimized
Only functions that can be used in Steady State may be used for non-optimized tag parameter expressions. These expressions are re-evaluated whenever any of the parameter values change. You cannot use Script-only functions in this situation.

Page Code, Services, Reports, etc.
These are full VTScada modules, declared in the application's AppRoot file. The full VTScada language and function list can be used.

Format Examples for Functions

The format example, provided for every function, also provides relevant information about how to use the function and the library that the func-

tion is a part of. The indication of the library is especially important to anyone writing a Script-layer based application.

Optional Parameters

For most function examples, some of the parameters will be shown inside square brackets. These parameters are optional. If the default values, as described in the parameter descriptions, will serve for your purpose, then you may leave the parameters out. If you want to specify some of the optional parameters, then you must provide all the parameters between the last one required and the optional parameter you want to specify. Use Invalid for each of the intervening optional parameters that you do not want to specify.

Examples:

```
\System\DropList(X1, Y1, X2, Y2, Data, Title, Index, FocusID, Trig-
ger, NoEdit, Init, Variable [, DrawBevel, VertAlign, AlignTitle,
Style, BGColor, FGColor]);
```

All the parameters from DrawBevel onward are optional and may be left out of the function call. Assuming that valid values have been defined for the required parameters, this function will work if used as follows:

```
\System\DropList(X1, Y1, X2, Y2, Data, Title, Index, FocusID, Trig-
ger, NoEdit, Init, Variable);
```

If you wanted to draw a drop list with an orange background, and did not care about any of the other parameters, you could use:

```
\System\DropList(X1, Y1, X2, Y2, Data, Title, Index, FocusID, Trig-
ger, NoEdit, Init, Variable, Invalid, Invalid, Invalid, Invalid, 135)
```

Will this Function Work in a Script Application?

Most development work is done within standard applications (those based on the VTScada layer), and the documentation is written from that point of view. Script applications will not have access to function libraries that were created explicitly for the VTScada layer. Do not assume that any function will work in your script application until you have tested it. It is possible to offer general guidelines for recognizing which functions are likely to work in a script application, but again, you should always

test first. When testing the function in a script application, try first with the format as shown. If the test fails, try using the function with the prefix \Layer\.

- Functions that are part of the \System layer will work in script applications.
- Most, but not all, of the basic string handling, math, time and date functions will work in a script application.
- If the format example begins with a backslash (\) and is not part of the \System layer, then it is likely that the function will not work in a script application, but test to be sure.

Obsolete Functions

Functions that are obsolete in VTScada version 11.2 are listed here. In most cases, code containing these functions will continue to compile, but the function call will do nothing.
If your legacy application uses any of the these functions, please refer to the documentation matching your software for details. Otherwise, they should be removed from all code intended for the current release.

ActiveMonitor (Alarm Manager)
AddPageToApp
Alarm
AlarmCat
AlarmInst
AlarmSoundCheck
AlmAck
AlmAckID
AlmArray
AlmCatName
AlmColor
AlmEnable
AlmList
AlmTone

AnimateState
CollapseTree
Cut
DeleteListItem
DoAcknowledge (Alarm Manager)
DragState
GetClientsListed
GetLogHeader
GetSubGraphic
HighlightState
HighlightTree
InsertListItem
LastSelectedModule
LastSelectedState
Load
MakeTypeInstance
ModuleCollapsed
ModuleTree
ModuleTreeSize
MoveSelState
MoveState
NewPage
NumAlarm
OperationalChange
PackData
Palette
PickModule
PickState
ReadNum
ReadText
SelectStates
SetLogHeader
SetShelved (Alarm Manager)
SetStateColor

ShadowTree
ShelvedEvent
SpeechEnum
SpeechLexiconDlg
SpeechReset
SpeechSelect
SpeechSpeak
SpeechStream
StartSound (Alarm Manager)
StateDiagram
StateHighlighted
Table
ValidateHistory
VarAttributes

4BtnDialog

(System Library)

Description: Draws a message dialog with up to 4 buttons and 3 lines of text and returns the number of the button that was pressed.

Returns: Numeric

Usage: Steady State only.

Function Groups: Graphics

Related to: Window

Format: \System\4BtnDialog(Icon, Btn1Label [, Btn2Label, Btn3Label, Btn4Label, Text1, Text2, Text3, Open, Modal, BoldFirst, Title, XPosPtr, YPosPtr, CloseButton, DefFocus, HelpFileName, HelpContextID])

Parameters:

Icon

Optional. Any image value for the (41 x 41) icon to be displayed in the dialog. Set to "Invalid" to display only text with no icon.
predefined icons include: \System\Question_icon, \System\Error_icon, and \System\Warning_icon.

Btn1Label

Required. Any text expression for the label on the first button.

Btn2Label

An optional parameter giving any text expression for the label on the second button. If invalid, the button will be omitted.

Btn3Label

An optional parameter giving any text expression for the label on the third button. If invalid, the button will be omitted.

Btn4Label

An optional parameter giving any text expression for the label on the fourth button. If invalid, the button will be omitted.

Text1

Required. An optional parameter giving any text expression for the first line of text to be displayed.

Text2

Required. An optional parameter giving any text expression for the second line of text to be displayed.

Text3

Required. A parameter giving any text expression for the third line of text to be displayed. Must be given as Invalid if there is to be no third line.

Open

An optional parameter that is any logical expression. If true (non-0) the dialog will be open; if false (0), it will be closed. The default is true.
Open will be set to false once the dialog has been acknowledged.

Modal

An optional parameter that is any logical expression. If true (non-0), the dialog is modal, if false (0) it is non-modal.
This parameter will override the System NoModal flag; if it is omitted, the NoModal flag will prevail. If the NoModal flag is also omitted, the dialog will default to being modal.

BoldFirst

An optional parameter that is any logical expression. If true (non-0) the first line of text in the dialog will appear in boldface type. If false (0), it will not be bolded. The default is false.

Title

An optional parameter that is any text expression that gives the title to appear in the window's title bar. If this parameter is omitted or is invalid, the window (dialog) will not have a title bar and will not be repositionable by the user.

XPosPtr

An optional parameter that is a pointer to a variable in which the x-coordinate of the center of the dialog is stored.

YPosPtr

An optional parameter that is a pointer to a variable in which the y-coordinate of the center of the dialog is stored.

CloseButton

An optional parameter that is the index of the button to which the title bar close button should be mapped (from 1 to 4).
If CloseBtn is invalid, then the title bar close button is mapped to the first button with the same text as the CancelLabel or CloseLabel application properties.
If no buttons match these labels, then "0" is returned when the dialog is closed using the title bar close button.

DefFocus

An optional parameter that is the index of the button that should have the focus when the dialog opens (from 1 to 4). If DefFocus is invalid, the first button has the focus when the dialog opens.

HelpFileName

An optional parameter that is the name of the help file containing the help topic for this dialog (as identified in the HelpContextID parameter). If HelpFileName or HelpContextID are invalid, then the default is as per the Window function.

HelpContextID

An optional parameter that is the numeric context ID identifying the help topic for this dialog (within the help file identified in the HelpFileName parameter).
If HelpFileName or HelpContextID are invalid, then the default is as per the Window function.

Comments

This module is a member of the System Library, and must therefore be prefaced by \System\, as shown in the "Format" section. If you are developing a script application, use "System\..." rather than "\System\..." in the function call.
The return value for this module is in the range of 1 to 4 and identifies which button has been pressed. If one of the buttons is labeled "Cancel", pressing <ESC> will cause the

number of the "Cancel" button to be returned.
For any optional parameter that is to be set, all optional parameters preceding the desired one must be present, although they may be invalid.

Example:

```
If Valid(FileName) && ! Valid(OK);
[
  Size = FileFind(FileName, 8, 4);
  IfElse(Size[0], Execute(
         OpenErrDlg = 1,
         OK = 0
        ),
  { else }
     OK = 1
  );
]

System\4BtnDialog(MakeBitmap("Warning", 7) { Icon to use },
                              "OK", "Cancel", Invalid, Invalid
                              { 2 buttons },
                              "File exists !" {1st line of text},
                              "Replace it ?"  {2nd line of text},
                              Invalid         {3rd line of text},
                              OpenErrDlg { Opening condition });
```

This code waits until a file name has been set, presumably from an edit field or some other method not shown here, and then checks to see if it already exists in the current directory. If it does exist, an error dialog with "OK" and "Cancel" buttons appears, warning of the existence of the file.

A Functions

The sections that follow identify all VTScada functions beginning with "A".

ABS

Description: Returns the absolute value of a numeric expression.

Returns:	A positive, numeric value
Usage:	Script or steady state.
Function Groups:	Generic Math
Format:	Abs(X)
Related to:	Min | Max | Step
Parameters:	X Required. Any numeric expression.
Comments:	If the expression for X is positive, the value is returned as positive. If the expression for X is negative, the value returned is the negative of the expression (i.e. the returned value is always positive or "0")

Example:

```
Angle = Abs(encoderAngle);
```

Related Functions:

AbsTime

Description:	Absolute time. This function returns true when a fixed time has been reached.
Returns:	Boolean
Usage:	Steady State only.
Function Groups:	Time and Date
Related to:	DateNum | Day | Month | Now | RTimeOut | Seconds | SetClock | TimeOut | TimeArrived | Today | Year
Format:	AbsTime(Enable, Interval, Offset)
Parameters:	***Enable*** Required. Any numeric (Boolean) expression giv-

ing the condition that allows the timer to operate.

When this parameter is true (i.e. not 0), the timer is "running". When this parameter is false (i.e. 0), the timer stops and the function has a value of false.

Interval

Required. Any numeric expression giving the time in seconds between the absolute time periods. This parameter must be strictly greater than 0, otherwise the function value will be invalid.

Offset

Required. Offset is any numeric expression giving the time in seconds to shift the absolute time from multiples of the Interval time.

Comments: The AbsTime function is designed to allow events to be scheduled at regular time intervals in real time. It is reset automatically when it appears in a true action trigger, or when it appears in a function that resets its parameters.

The parameters are relative to local time. If attempting to schedule an event for the same time across servers in multiple time zones, the offset should be an expression relative to each time zone.

Note: Note that the behavior will differ depending on whether you use this function in a script code module or in a tag expression. In script code, the function will be reset as described, and will wait for the next trigger to occur.

In a tag expression, this function will not be reset after triggering.

Since AbsTime is ultimately based on midnight Janu-

ary 1st, 1970 (a Thursday), it is possible in theory to use it for a weekly time interval, with an appropriate offset for the day of the week starting from Thursday. In practice, TimeArrived may be more appropriate for a weekly trigger.

Example:

```
If AbsTime(1, 86400 { 24 hrs }, 28800 { shifted 8 hrs });
[
...
]
```

In the example above, the script will be executed every 86400 seconds (24 hours), offset by 28800 seconds (8 hours) from midnight. This means that the trigger will become true at 8:00 AM every morning when the script executes. The function will then be reset to wait until 8:00 AM the following day to execute again.

```
If AbsTime(1, 3600 { 1 hr }, 0 { not shifted });
[
...
]
```

The example displayed above enables a user to schedule an event to run on the hour, every hour.

Related Information:

See: "Latching and Resetting Functions" in the VTScada Programmer's Guide

Accumulate

(Hierarchical Accumulator module)

Description: Adds a tag and a value to be counted for that tag, to a named accumulator.

Returns: Invalid

Usage: Script or steady state.

Function Groups: Variable

Related to: GetValue | GetContainerNumActive | GetContainerNumUnacked

Format: \ HierarchicalAccumulator\Accumulate(TagObj, AccumulatorName, Value [,LocalUniqueID]);

Parameters:

TagObj

Required. The tag object that is contributing to the count

AccumulatorName

Required. The name of the accumulator that the tag is contributing to. For example, two accumulator names in use by the Alarm Manager are, "AlarmUnacked" and "AlarmActive". If creating your own accumulator, you may use any name you wish.

Value

Required. The current value to count for the TagObj\LocalUniqueID. Commonly a 1 or 0.

LocalUniqueID

Optional. If one tag contains item to be accumulated, use this field to provide a locally unique identifier for each.
Example: for accumulation of active alarms in Analog Status tag, this parameter will be "Hi" or "Lo", depending on which built-in alarm is being counted.

Comments: The Accumulate function always returns Invalid. It may be called in steady state (for concise code), or as a subroutine call in a script in order to save RAM.
This function is part of the HierarchicalAccumulator module, so must always be called as shown in the format. You will need this function if creating a new type of tag, which also contains its own built-in alarms. You may create an accumulator for any property of your tag.

The accumulator enables a fresh count to be generated at different levels in a tag tree, and as tags are moved or disabled.

Access to the accumulation of active and unacknowledged alarms is provided by the Alarm Manager's GetContainerNumUnacked and GetContainerNumActive functions. To access accumulations of your own creation, use the \HierarchicalAccumulator\GetValue function.

Examples:

```
If Watch(1, AlarmLoUnacked);
[
   \HierarchicalAccumulator\Accumulate(Root, "AlarmUnacked",
AlarmLoUnacked != 0, "Lo");
]
```

Related Functions:

Acknowledge

Note: Deprecated. Do not use in new code.

(Alarm Manager module)

Description: Will acknowledge an alarm.

Returns: 0

Usage: Script Only.

Function Groups: Alarm

Related to: CurrentTime | Register (Alarm Manager)

Format: \AlarmManager\Acknowledge(AlarmName[, EventTime, Operator]);

Parameters:

AlarmName

Required. The name of the alarm (not the alarm object value that was passed to the Register subroutine) that

will be acknowledged

EventTime

Optional. Timestamp to use when adding this event to the alarm lists. Defaults to CurrentTime()

Operator

Optional text. The name of the operator who acknowledged the alarm.
If invalid, the logged-in user (according to the Security Manager) will be used. If the Security Manger user is empty, then the application property, "LoggedOffLabel" will be used.

Comments: This subroutine's primary responsibility is to send RPC messages to other workstations. DoAcknowledge does the actual work of acknowledging. When an acknowledgment is done, the only field that will change is status. To change any other field, a custom DoAcknowledge that is aware of the structure of alarms within the application must be written.

ACos

Description: Calculates the trigonometric arc cosine in radians.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Trigonometric Math

Related to: ASin | ATan | Cos | Sin | Tan

Format: ACos(X)

Parameters:

X

Required. Any numeric expression in the range -1 to +1.

Comments: The returned angle is in radians. To convert an angle from radians to degrees, divide by \pi / 180 or (approximately) 0.0174533.

Example:

```
radAngle = ACos(0);
degAngle = radAngle / \pi / 180;
```

In the example shown above, the value of degAngle will be 90.

AcquireLock

Description: Subroutine to acquire an exclusive lock on reading/writing working copy files across all applications.

Returns: Semaphore

Usage: Script Only.

Function Groups: Configuration Management

Related to: ReleaseLock |

Format: Layer\Acquirelock(WritePermission, Owner, SemaphorePtr, WrtingUserID[, IgnoreState])

Parameters:

WritePermission

Required. Any BOOLEAN expression. Set TRUE if the owner intends to perform writes or FALSE for read-only access.

Owner

Required. The instance of the lock owner.

SemaphorePtr

Required. A pointer that will be set TRUE when the semaphore is granted.

WritingUserID

Optional. UserID (from Layer\GetUserID) for the user

acquiring the lock. Should be used if the user is going to make working copy changes and not commit them within the lock, so that the correct UserID is assigned to the changes when they are committed.

IgnoreState

Optional. Set true to ignore activation. Defaults to FALSE.

Comments: This function should be called before performing any modification of any application's working copy. The output of this function is a semaphore that guarantees exclusive access to the working copy until it is released.
Note that the caller must wait for the semaphore to become TRUE (1), which indicates that the lock is held.
A lock gathered by AcquireLock should be released by calling ReleaseLock. It is automatically released when the variable used to collect the semaphore is released during calling object destruction. This module launches a destructor module so that if the caller to this module stops, the destructor will continue to run until ReleaseLock is called or one of the parameters becomes invalid.
Since this lock prevents any other access to the working copy, you should minimize the length of time it will be held.

Examples:

```
GetLayerLock [
  If 1 ProcessFiles;
  [
    AcquireLock(1 {write}, Self, &LayerLock);
  ]
]
```

Active

Note: Deprecated. Do not use in new code.

(Alarm Manager module)

Description: Tells the Alarm Manager to activate an alarm. This subroutine will activate an alarm and signal it as unacknowledged.

Returns: 0

Usage: Script Only.

Function Groups: Alarm

Related to: Register (Alarm Manager) (Alarm Manager) | CurrentTime | Normal | IsActive

Format: \AlarmManager\Active(AlarmObject[, EventTime]);

Parameters:

AlarmObject

Required. Object Value for the alarm that was passed to the active Register subroutine.

EventTime

Optional, Timestamp to use when adding this event to the alarm lists. Defaults to CurrentTime()

Comments: The Active subroutine always returns "0".

Examples:

To avoid an IF 1 condition when activating an alarm, it is common practice to include a variable to ensure that the script runs only once. This should take its value from the current alarm state.

```
Init [
   AlarmOn = AlarmManager\IsActive(MyAlarm);
]

Main [
   IF value >= SomeSetPoint && ! AlarmOn;
[
   AlarmOn = 1;
```

```
   AlarmManager\Active(MyAlarmObj);
]
```

ActiveCode

Description: ActiveCode returns the code value of the active statement in the given module instance.

Returns: Code Value

Usage: Script or steady state.

Function Groups: Compilation and On-Line Modifications, Advanced Module

Related to: ActiveState | ActiveWindow | CurrentWindow | StateName | ModuleFileName

Format: ActiveCode(Object)

Parameters:

Object

Required. Any object expression to be monitored in the module instance.

Comments: A code value is a combination of module, state and statement that is the active statement in the given module instance (object value).

Example:

```
stmt = ActiveCode(readObj);
```

ActiveState

Description: ActiveState returns the code value of the active state in the given module instance.

Returns: Code Value

Usage: Script or steady state.

Function Groups: Compilation and Online Modifications, States

Related to: ActiveCode | ActiveWindow | CurrentWindow | StateName | ModuleFileName

Parameters:

Object

Required. Any object expression to be monitored in the module instance.

Comments: A code value is a combination of module and state that is the active statement in the given module instance (object value).

Example:

```
state = ActiveState(readObj);
```

ActiveWindow

Description: ActiveWindow returns the object value of the root module instance in the current active window.

Returns: Object value

Usage: Script or steady state.

Function Groups: Compilation and Online Modification, Window

Related to: ActiveCode | ActiveState | CurrentWindow | Window

Format: ActiveWindow( )

Parameters: None

Comments: Child windows (those with bit 9 set in their Window call) are not recognized as separate entities; clicking on a child window returns the object value of the root module in its parent window. This is not true for owned windows (those with bit 15 set in their Window call), which return the object value of the root module instance in the window.

Example:

```
object = ActiveWindow();
```

ActiveX

Description: Instantiates an ActiveX object. An ActiveX object is treated as a COM client interface that requires a client window area in which to draw.

Usage: Steady State only.

Warning: This function should be used only by advanced programmers. It may interfere with VTScada graphics.

Function Groups: COM

Related to: COMClient | COMEvent | COMStatus

Format: ActiveX(X0, Y0, X1, Y1, ObjectIdentifier [, EventSearchScope, EventParent, EventCaller, LicenseString])

Parameters:

X0, Y0, X1, Y1

Required These are the numeric coordinates of the client window area in which the COM object is to draw itself.

ObjectIdentifier

Required. Specifies a unique identifier for the object that is to be instantiated. It may take one of the following forms:

• A text string representing a ProgID (e.g. "Excel.Application").

• A textual GUID in registry format (e.g. "{000208-0000-0000-C000-000000000046}"). Note that the curly braces are compulsory.

• A binary GUID (e.g. the result from "GetGUID(1, 00020812-0000-0000-C000-000000000046)").

EventSearchScope

Optional An object value that, if present, specifies the scope in which to search for event subroutines.
May be any expression that yields a module value or object value. Defaults to Self().

EventParent

Optional Any expression yielding an object value. If present, specifies the context that is used to resolve scope for event subroutines. May be any expression that yields an object value.
Defaults to Self().

EventCaller

Optional. Any expression that yields an object value. If present, this specifies an "auxiliary" context for event subroutines. An event subroutine can retrieve this value using Caller(Self())
Defaults to Self().

LicenseString

Optional. Used to provide a license key that will be passed to the ActiveX control when instantiated, permitting the use of ActiveX controls that require license data for activation. This will work for both server-instantiated ActiveX controls and those that are remotely called by a VIC.

Comments:

If the statement succeeds, a COM client interface is returned, allowing subsequent access to the object. If it fails, Invalid is returned.
There are two significant parametric differences between an ActiveX function and a COMClient function. Firstly, the ActiveX function requires a client window in which to draw. Secondly, there is no ObjectContext parameter. ActiveX objects are only instantiated in process, as they require direct GDI access to process resources [such as the client window area].
A window in VTScada acts as a container for ActiveX objects, in the true OLE definitions of the OC96 specification. This architecture provides a container

enumerator so that an ActiveX object can interact with other ActiveX objects in the same container. Like the COMClient function, this function returns an opaque COM Client Interface handle, through which subsequent object manipulations are performed.

Unlike the COMClient function, this statement may be only be used as a steady-state statement.

The ActiveX object will only remain instantiated while the steady-state statement is still running (i.e. a change of state or destruction of the module instance which is running the statement will cause the ActiveX object to be destroyed). Any variables that hold a handle to the COM Client Interface will be invalidated at that time.

Note: Within an Anywhere Client session, this function does nothing.

Example:

```
[
Loaded = 0;
BrowserObj;
]
BrowserObj = ActiveX(12, 52, w - 22, h - 42, "Shell.Explorer");
If Timeout(Valid(BrowserObj) && !Loaded, 1);
[
BrowserObj\Navigate("www.google.com");
Loaded = 1;
]
```

Example 2:

```
MyObject = ActiveX(… );
X = MyObject\DoSomething(aValue); { A method call on the object }
MyAxObjects\AProperty = "Hello"; { A property set }
Y = MyAxObject\AnotherProperty; { A property get }
```

AddAccount

Security Manager Module

Description Creates a new account (either a user account or a role).

Returns Object value

Usage Script Only.

Related to: ModifyAccount | DeleteAccount

Format \SecurityManager\AddAccount (NewAccountData [, PtrReturnCode, HaveLock]);

Parameters

NewAccountData

Required. An AccountData structure, a single dimension array of AccountData structures or a dictionary of AccountData structures containing the data for the new account(s).

PtrReturnCode

Optional. A pointer to a value that will contain one of the defined result codes at the conclusion of the operation.

HaveLock

Optional. A Boolean value that indicates whether the working copy lock is held by the calling code. Default FALSE.

Comments To use this API, the calling code must be running in a security session that has Manager privilege.

Adding an account is an asynchronous operation. If the asynchronous operation was not attempted, due to detection of an error, the return value will be Invalid. If the asynchronous operation is attempted, the return value will be an object value. The object value will become Invalid when the asynchronous operation completes. At that time (or when the method returns Invalid), the value addressed by

PtrReturnCode can be examined to determine the status of the operation. The contents of the value addressed by PtrReturnCode are undefined until the method returns Invalid.

A single account can be added by supplying a single AccountData structure in NewAccountData. Multiple accounts can be added in one operation by providing a single dimension array or dictionary of AccountData structures in NewAccountData.

The result code returned in the value addressed by PtrReturnCode will be a scalar value if a single structure was supplied in NewAccountData. If an array of structures or a dictionary of structures was supplied, a single dimension array of the same size as NewAccountData will be returned in the value addressed by PtrReturnCode, each element containing the result code for the corresponding NewAccountData element.

Adding an account requires a working copy write lock. If such a lock is held by the calling code, the HaveLock parameter must be set to TRUE. Otherwise omit this parameter or set it to FALSE. If the calling code holds a read lock on the working copy, this must be released before AddAccount can complete its operation.

The AccountData structure(s) provided must have the Username member set to a unique name. If the account being added is a user account (as opposed to a role), a password conformant with application password strength settings must be provided.

On return the AccountID member of each AccountData structure that was successfully processed will be set to a

unique user ID that will not change for the life of the account. The Password member is not erased. It is highly recommended that calling code be careful to ensure that unencrypted passwords are destroyed as soon as possible after completion of this operation.

AddConnection

(ODBC Manager Library)

Description: Called to get the object value of an ODBC connection. If the connection does not exist, then a new ODBC DSN connection will be created and explicitly added to the ODBC Manager's internal list of DSN's

Returns: The object value of an ODBC connection

Usage: Script or steady state.

Function Groups: ODBC

Related to:

Format: \ODBCManager\AddConnection(DSN[, UserName, Password, ConnectTimeout, ExecutionTimeout, IdleTime, CheckBackupTime, DisconnectOnError, HandleError, UseDriverTimeout, MirrorToDisk, DBType, PaceRecovery, PaceRecoveryRate, RollbackOnConnect, ReconnectDelay])

Parameters:

DSN

Required. Text identifying the database to connect to. This can be a DSN (data source name) or it could be a connection string containing a fileDSN parameter such as: "filedsn = C:\VTScada\Access.dsn; dbq = c:\testdb1.mdb"

UserName

Optional. An optional parameter providing the user name for database access.

If both the Username and Password are missing from the function call, a dialog box will prompt the operator to provide values for these parameters.

Password

Optional parameter for providing the password, if required, for database access. See note for UserName.

ConnectTimeout

Optional numeric parameter specifying the length of time the module will wait for a valid database connection to be made.
Defaults to 30 if missing or invalid

ExecutionTimeout

Optional parameter specifying in seconds, the length of time the method will wait for a database command to execute. The parameter, UseDriverTimeout must be true for this parameter to have any effect.
Defaults to 60.

IdleTime

Optional parameter specifying in seconds, the length of time will wait with no database communication before closing the connection.
Defaults to 600 if missing or invalid

CheckBackupTime

Optional numeric parameter specifying the length of time between checks for cache files left behind failed log attempts.
Defaults to 300 if missing or invalid

DisconnectOnError

Optional Boolean parameter indicating whether the ODBC connection should be terminated after any execution error occurs.
Defaults to FALSE (0) if missing or invalid.

HandleError

Optional parameter that can be any of:

- an error handle object,
- an array of errors,
- a Boolean value. TRUE to disconnect or false to indicate no disconnect.

Defaults to false, meaning 'no disconnect'.

UseDriverTimeout

Optional Boolean value which may be set to true if ODBC driver in use supports timeouts.
If so, then the execution timeout limit (ExecutionTimeout) is handed to the ODBC functions.
Defaults to FALSE (0) if missing or invalid.

MirrorToDisk

Optional Boolean value. If true, this specifies that every write should go to disk.
Defaults to FALSE (0) if missing or invalid.

DBType

Optional numeric value, indicating the type of this DB connection. Defaults to 0 if missing or invalid.

DBType	Meaning
0	MS SQL
1	MS Access
2	Oracle
3	MySQL
4	SyBase

PaceRecovery

Optional, Boolean value. Set to TRUE (non-zero) to pace the recovery of logs using the value in the following parameter.
Defaults to TRUE (1) if missing or invalid.

PaceRecoveryRate

Optional. Any numeric expression that will set the time in seconds for recovery of logs. The preceding parameter, PaceRecovery must be set to true.
Defaults to 0.1 if missing or invalid.

RollbackOnConnect

Optional. Any Boolean expression. If set to TRUE (non-zero) a rollback will be done on connect.
Defaults to TRUE (1) if missing or invalid.

ReconnectDelay

Optional numeric expression to set the time to wait for reconnect on a failed connect.
Defaults to 5 if missing or invalid.

Comments: This module is a member of the ODBCManager Library, and must therefore be prefaced by \ODBCManager\, as shown in "Format" above.
The only way to ensure that a long running (or faulty) query does terminate is to set appropriate values for both the parameters ExecutionTimeout and UseDriverTimeout.

AddContributor

Description: Adds a contributor to a container.

Returns: Nothing

Usage: Script Only.

Function Groups: Containers and Contributors

Related to: DeleteContributor | GetContributors | PContributor |

Format: AddContributor(HandleName, ArrayName, CountName, ContainerObj, ContributorObj, IndexAddress, Value, CountIncrement);

Parameters:

HandleName

Required. Any expression that evaluates to the name of the handle variable in the container module.

ArrayName

Required. Any expression that evaluates to the name of the variable in the ContainerObj parameter, that holds an array of values to which the contributor is to be added.

The ArrayName parameter may be invalid if there is no such array in the container.

CountName

Required. Any expression that evaluates to the name of the variable in the ContainerObj parameter, that holds a count of the current number of this type of contributor.

CountName may be invalid if no such variable exists in the ContainerObj. Not all contributors need to be counted.

Note that CountIncrement determines the initial change in the count and the contributor must maintain the count.

ContainerObj

Required. Any expression that evaluates to the object value of the container tag module.

ContributorObj

Required. Any expression that evaluates to the object value of the new contributor to add to the container.

IndexAddress

Required. Any expression that evaluates to the address pointer of the variable holding the contributor index.

Value

Required. Any expression that evaluates to the current

numeric value to set in the container's ArrayName array.
Value may be Invalid. Value may also be updated at any time by the contributor by scoping into the ArrayName array in the container, and setting the array element at the index that will be set in the variable pointed to by IndexAddress.

CountIncrement

Required. Any expression that evaluates to the numeric value that will be added to the variable in the container that has the same name as CountName. CountIncrement's value is usually a "1" or a "0", indicating whether or not the contributor is actively contributing its value. The contributor will increment or decrement the value of the CountName variable as the corresponding state of the contributor changes.

Comments: This function can be called from the contributor.

AddEditorText

Description: Inserts a text string into a text editor.

Returns: Nothing

Usage: Script or steady state.

Function Groups: Editor

Related to: CurrentLine | Editor | ForceEvent | GoToOffset | MakeEditor | SetEditMode

Format: AddEditorText(Editor, Text)

Parameters:

Editor

Required. Any expression that evaluates to an editor value that was created by the MakeEditor function.
If this is not an editor type value, then the function will

do nothing.

Text

Required. Any expression that evaluates to the text to insert into the text editor.

Comments: The text will be broken into lines based on carriage returns or line feeds or both.

Example:

```
AddEditorText(AlarmLog, "Reason for alarm:");
```

This example inserts the text "Reason for alarm" into the text editor identified in the variable AlarmLog.

AddModule

Description: Adds a new module to an existing module and returns the value of the newly created module.

Warning: This function may cause irrecoverable alteration of your application. It should be used only by advanced programmers.

Returns: A new module value

Usage: Script Only.

Function Groups: Compilation and On-Line Modifications, Advanced Module

Related to: AddOptional | AddParameter | AddState | AddStatement | AddVariable | CreateModule | FirstState

Format: AddModule(Parent, Name, Reserved, Attrib, Class, VarTextSize, DocFileName)

Parameters:

Parent

Required Any expression that returns a module value. Specifies the parent module of the new module.

Name

Required. Any text expression providing a name for the new module.

Reserved

Reserved for future use. Should be set to 0.

Attrib

Required. Any numeric expression giving the module's attribute bits as follows:

Attrib	Bit No.	Attribute
0	-	No special attributes
1	0	Reserved – set to 0
2	1	Queued
4	2	Reserved – set to 0
8	3	Reserved – set to 0
16	4	Reserved – set to 0
32	5	Reserved – set to 0
64	6	Reserved – set to 0
128	7	Reserved – set to 0
256	8	Reserved – set to 0
512	9	Protected

Class

Required. Any numeric expression in the range 0 to 65535, giving the class for the new module.

VarTextSize

Required. Any numeric expression giving the length of the variable declaration for the new module (i.e. NewMod MODULE { a new module };).

DocFileName

Required. Any text expression giving the file name of

the new module's definition document.

Comments: None.

AddOptional

Description Adds a new statement to an action script and returns its own error code.

Warning This function may cause irrecoverable alteration of your application. It should be used only by advanced programmers.

Returns A new module value

Usage Script Only.

Function Groups Compilation and Online Modifications, States

Related to: AddVariable | GetStatement

Format AddOptional(NewStatement, Destination, Location, TextSize)

Parameters

NewStatement

Required. Any expression that returns a statement value. This is the value of the new script statement to be added.

Destination

Required. Any code value expression. Destination is the action that will receive NewStatement.

Location

Required. Any numeric expression giving the location of NewStatement within the script.

TextSize

Required. Any numeric expression giving the length of NewStatement's text in characters.

Comments The function returns "0" if successful and a non-0 value if it

fails. AddOptional is disabled in the run time version of VTS; it will do nothing and returns Invalid.

AddParameter

Description: Adds an existing variable as a module parameter and returns the number of parameters in the module.

Warning: This function may cause irrecoverable alteration of your application. It should be used only by advanced programmers.

Returns: Numeric... the number of parameters in a module.

Usage: Script Only.

Function Groups: Compilation and Online Modifications, Advanced Module

Related to: ConstCount | NumParms | SetOneParmText | SetParameter | SetParmText | AddVariable

Format: AddParameter(Variable, Position)

Parameters:

Variable

Required. Any expression that returns a variable value such as AddVariable(...).
This variable will become a parameter in the module where it is defined.

Position

Required. Any numeric expression giving the position in the parameter list where Variable will be inserted.
Position 1 is the first parameter.
Inserting a variable past the end of the list will append the variable to the parameter list.

Comments The return value of AddParameter is the number of parameters in the module after the addition, or "-1" if it failed.
Caution: Attempting to insert a variable in the parameter list twice may result in undefined behavior.

AddPrivToUser

(Security Manager Library)

Description: A subroutine to grant a privilege to a user.

Returns: Numeric (via the first parameter)

Usage: Script Only.

Function Groups: Security

Related to:

Format: \SecurityManager\AddPrivToUser(PtrReturnCode, User-name, Privilege[, HaveLock])

Parameters:

PtrReturnCode

Required. A pointer to a variable that will be used for the return code.

PtrReturnCode	Meaning
1	Privilege added
2	Denied. The calling context does not have the Manager sys-tem privilege.
3	Privilege is not valid. No action taken.
4	The user does not exist. No action taken,
6	The application cannot be edited.

UserName

Required. Any expression for the name of the user account to create.

Password

Required. Any expression for the password of the user account to create.

Privilege

Any numeric expression for the privilege to grant to the user. Set the value negative for system privileges and positive for application privileges.

AppPrivileges

Optional. Any numeric expression for the bit-wise set of application privileges. If missing or invalid, no application properties are granted.

AltID

Optional. Any expression for the alternative logon identifier.

AutoLogoff

Optional. Any numeric expression for the user-specific logoff time (specified in seconds).

PWDate

Optional. The password creation date in days since January 1, 1970.

HaveLock

Optional Boolean expression. Set to true if we have the WC lock. Defaults to 0 or FALSE.

Comments: May only be called from a user-context that has the Manager system privilege. The return value of the function is the object value of the launched worker module. This will be set to Invalid when the operation has completed and may be used to discover when that occurs.
Use of this function requires an understanding of the VTScada security system and the system privileges. Please refer to System Privileges in the chapter Security Manager Service.

AddRead

Description: Add a read request.

Returns: Nothing

Usage: Script Only.

Function Groups: Memory I/O

Related to: DelRead

Format: VTSDriver\AddRead(Address, N, Value, Rate[, OriginalAddr, DisablePolling)

Parameters:

Address

Required. Any expression for the address or array of text addresses from which to get the data.

N

Required. Any expression for the number of elements to retrieve.
If Address is an array, N should be the number of items in the arrays.

Value

Required. A pointer to the destination for the read data. If Address is an array, Value should also be an array of pointers to the data destination.

Rate

Required. Any numeric expression for the update rate, measured in seconds.

OriginalAddr

For use only by the Driver Multiplexer. Set Invalid in all other instances. The original address string specified in the I/O tag before being parsed into an value that is fit for the Address parameter of AddRead. For example, the original address may be {40001}{42001},

but the address given to the subordinate driver would be 40001.

DisablePolling

Primarily for use by the Driver Multiplexer. In general, should be set to Invalid for other drivers. Any numeric expression, setting a rate value for which Polling will be disabled. Polls will then only be performed by a call to Driver\PollAll() or by setting the Read module's Trigger.

Comments: This module is called by a tag to add a request to read a specific range of memory and set the resulting read data into the variable pointed to by the third parameter. The pointer may be a simple variable for one element read, an array if more than one element is requested, or an object reference. Use of an object for the Value parameter, rather than an array, allows a synchronous way of reporting new data. The object must be a module with a subroutine named NewData. NewData will have the following parameters:

Address

Required. The original address that AddRead was called with. Should be an array if Data is an array.

TimeStamp

Required. The timestamp of the data, in UTC. Should be an array if Data is an array.

Data

Required. Any numeric expression for the update rate, measured in seconds. May be an array of arbitrary size.

Attribute

Optional. Auxiliary data value - rarely used.

ServiceSync

Optional Boolean. TRUE when NewData is being called because the service is synchronizing, rather than when the driver reported new data.

The NewData callback may only be called if RefreshData resulted in a value change (affected by PropagateOnlyOnDataChange). If Data is an array, and only element [i] changed, then the TimeStamp array is used to indicate changes. A valid TimeStamp[i] means that Value[i] and Attribute[i] are valid. Invalid TimeStamp[i] means that Value[i] and Attribute[i] will be invalid since they have not changed.

Example:

```
{ SetDefaultSetpoints }
(
)
[
  Var;
]
Init [
  If (Variable("PumpStationSimulator")\Ready == 1) Main;
  [
    Variable("PumpStationSimulator")\Driver\Addread(40001, 1,
             &Var, 1);
  ]
]
Main [
  If Watch(0, Var);
  [
    ...
  ]
```

AddressEntry

Description: Checks whether the attached driver has an AddressAssist module and uses that if available. Otherwise, presents a standard edit field into which the I/O address may be entered.

Returns: Nothing

Usage: Steady State only.

Function Groups: Graphics

Related to: GUITransform | PAreaSelect | PCheckBox | PContributor | PDroplist | PEditfield | PPageSelect | PRadioButtons | PSecBit | PSelectObject | PSpinbox | PTypeToggle

Format: \DialogLibrary\AddressEntry(X1, Y1, X2, Y2, Var, IODevice, SupportedData, FunctionType, Title, FocusID[, Trigger, DrawBevel, BGColor, FGColor])

Parameters:

X1

Required. Any numeric expression giving the X coordinate on the screen of one side of the object and its label. The smaller of X1 and X2 will always be to the left

Y1

Required. Any numeric expression giving the Y coordinate on the screen of either the top or bottom of the object. The smaller of Y1 and Y2 will always be the top.

X2

Required. Any numeric expression giving the X coordinate on the screen of the side of the object and its label opposite to X1.

Y2

Required. Any numeric expression giving the Y coordinate on the screen of the top or bottom of the object, whichever is the opposite to Y1.

Var

Variable to be set by AddressEntry

IODevice

Any expression for the name of the I/O device driver being used.

SupportedData

A bitwise expression, indicating the data type.

Bit	Meaning when set
0	Digital
1	Analog
2	Text

FunctionType

Any Boolean expression, indicating whether the function should be read (0) or write (1).

Title

Any text expression to use as the title for the field.

FocusID

Any numeric expression for the focus number of this graphic. If this value is 0, the field will display its current setting, but will not be able to be opened (i.e. its value cannot be changed), and will appear disabled (grayed-out).

Trigger

A parameter whose value is derived from ZEditField and can therefore be set to "0" (internal buffer changed), "1" ("Enter" key pressed), or "2" (focus lost). If this information is not required and the next parameter is used, a value of invalid or a constant may be substituted.

DrawBevel

Any logical expression. If TRUE, a bevel is drawn around the graphic.

BGColor

Optional. Any numeric expression for the background VTScada Color Palette of the control. No default value.

FGColor

Placeholder for the foreground color of the control. Not currently implemented.

Comments: none

AddState

Description: Adds a new state to an existing module and returns its state value.

Warning: This function may cause irrecoverable alteration of your application. It should be used only by advanced programmers.

Returns: State value

Usage: Script Only.

Function Groups: Compilation and Online Modifications, States

Related to: AddOptional | AddStatement | AddVariable | FirstState

Format: AddState(Module, Name, Reserved, Size)

Parameters:

Module

Required. Any expression that returns a module value.

Name

Required. Any text expression giving the name of the new state.

Reserved

Reserved for future use. Should be set to 0.

Size

Required. Any numeric expression giving the length of the state definition text, measured in characters.

Comments: None

AddStatement

Description: Adds a new statement to an existing state and returns its own error code.

Warning: This function may cause irrecoverable alteration of your application. It should be used only by advanced programmers.

Returns: Error code (0 = success, non-zero = failure)

Usage: Script Only.

Function Groups: Compilation and Online Modifications, States

Related to: AddVariable | MakeDAG | Compile

Format: AddStatement(Statement, Destination, TextSize)

Parameters:

Statement

Required. Any expression that returns a statement value. Commonly generated with the Compile function.

Destination

Required. Any expression that returns a code value. This indicates where to insert the new statement. If no statement is present in the code value, the new statement will be appended.

TextSize

Required. Any numeric expression giving the length of the statement text, measured in characters.

Comments: AddStatement doesn't affect the .SRC file; it affects the expected location of items in the .SRC file. Both must be updated in unison.
The return value is 0 if successful and non-zero if the function fails.
AddStatement is disabled in the run time version of VTS; it

will do nothing and return invalid.

AddUser

(Security Manager Library)

Description: Adds a user to the system, specifying name, password, privileges, etc.

Returns: Numeric (via the first parameter)

Usage: Script Only.

Function Groups: Security

Related to:

Format: \SecurityManager\AddUser(PtrReturnCode, Username, Password[, SysPrivileges, AppPrivileges, AltID, AutoLogoff, PWDate, HaveLock])

Parameters:

PtrReturnCode

Required. A pointer to a variable that will be used for the return code.

PtrReturnCode	Meaning
1	User added
2	Denied. The calling context does not have the Manager system privilege.
3	User already exists. No action taken.
6	The application cannot be edited.

UserName

Required. Any expression for the name of the user account to create.

Password

Required. Any expression for the password of the user account to create.

SysPrivileges

Optional. Any numeric expression for the bit-wise set of system privileges to be granted to the new user. If missing or invalid, no system privileges are granted.

AppPrivileges

Optional. Any numeric expression for the bit-wise set of application privileges. If missing or invalid, no application properties are granted.

AltID

Optional. Any expression for the alternative logon identifier.

AutoLogoff

Optional. Any numeric expression for the user-specific logoff time (specified in seconds).

PWDate

Optional. The password creation date in days since January 1, 1970.

HaveLock

Optional Boolean expression. Set to true if we have the WC lock. Defaults to 0 or FALSE.

Comments: May only be called from a user-context that has the Manager system privilege. The return value of the function is the object value of the launched worker module. This will be set to Invalid when the operation has completed and may be used to discover when that occurs.
Use of this function requires an understanding of the VTScada security system and the system privileges. Please refer to System Privileges in the chapter Security Manager Service.

AddVariable

Description: Adds a new variable to an existing module and returns its variable value.

Warning: This function may cause irrecoverable alteration of your application. It should be used only by advanced programmers.

Returns: Variable

Usage: Script Only.

Function Groups: Compilation and Online Modifications, Variable

Related to: DeleteVariable | MakeNonShared | MakeNonPersistent | MakePersistent | MakeShared | SetDefault | SetVariableClass | SetVariableText

Format: AddVariable(Module, Name, Reserved, Attrib, Class, PersistentSize, VarTextSize, NumDimensions, ArrayElem, ArraySize)

Parameters:

Module

Required. Any expression that returns a module value.

Name

Required. Any text expression that gives the name of the new variable.

Reserved

Reserved for future use. Set to 0.

Attrib

Required. Any numeric expression giving the variable attribute bits as follows:

Attrib	Bit No.	Attribute
1	0	Array
2	1	Shared

4	2	Persistent
8	3	Module
16	4	Parameter
32	5	Constant
64	6	(Obsolete) Reserved – set to 0
128	7	(Obsolete) Reserved – set to 0
256	8	Temporary *
512	9	Protected
1024	10	Variable is an instance variable (see comments)

A variable cannot be both persistent and temporary, since a persistent variable is stored on disk, and a temporary variable exists only while VTScada is running or until the application is recompiled.

* Note: Temporary variables should be used only by advanced users, since recompiling the application destroys them.

Class

Required. Any numeric expression in the range 0 to 65535, giving the variable class number for the new variable.

PersistentSize

Required. Any numeric expression giving the number of bytes of storage allocated in the .VAL persistent variable file for this variable.

For array types, set this to the byte size of the largest array element (normally 8 bytes for numeric values).

For arrays containing text, enter the character length of the longest string element.

Use a valid 0 if this isn't a persistent variable.

VarTextSize

Required. Any numeric expression giving the length of the variable declaration text in characters.
This parameter is ignored for temporary variable types.

NumDimensions

Required. Any numeric expression that gives the number of array dimensions for the variable.
NumDimensions should be "0" for a simple variable.

ArrayElem

Required. Any array element giving the starting element in the array. The subscript for the array may be any numeric expression.
If NumDimensions is 0, ArrayElem is ignored. I
f NumDimensions is 1, this specifies the starting index for the array.
If NumDimensions is greater than 1, this is element of an array of starting indices for each dimension of the multidimensional array.

ArraySize

Required. Any numeric expression specifying the size of the array.
If NumDimensions is 0, ArraySize is ignored.
If NumDimensions is 1, this specifies the number of elements in the array.
If NumDimensions is greater than 1, this is an array of sizes for each dimension of the array.

Comments:

This function doesn't affect the .SRC file; it affects the expected location of items in the .SRC file. Both must be updated in unison.
The return value is the new variable added. If the variable already existed, it will remain unchanged and the return value will be Invalid.
Bit 10 (1024) of the Attrib parameter specifies that the vari-

able is an instance variable. An instance variable is a temporary variable that gets added to a single instance of a module, rather than all instances of a module. If you specify this bit, then you must have passed an object (module instance) into the first parameter. Instance variables cannot also have the attributes Shared, Persistent, Module, or Parameter.

Example:

```
If 1 Main;
[
  IfThen(! Valid(FindVariable(varName, Self(), 0, 1))
  { Variable doesn't exist },
  AddVariable(Self() { Create variable as part of this module },
              varName { Name of new variable },
              0 { Reserved },
              1 { Variable is type array },
              0 { Class },
              0 { Variable not persistent },
             20 { Name is 20 chars long },
              1 { Var is 1 dimensional array },
              0 { Array starting index },
              220 { Number of elements in array }));
]
```

In the example above, the statement checks to see whether or not a certain (array) variable exists; if it does not, the statement creates a 1 dimensional array (named VarName) with 220 elements.

AdjustArray

Description:	Changes the array information for a variable.
Warning:	This function removes existing information from the array.
Returns:	Nothing
Usage:	Script Only.
Function Groups:	Array, Compilation and On-Line Modifications
Related to:	ArrayOp1 | ArrayOp2 | New
Format:	AdjustArray(Variable, NumDimensions, Start, Size)

Parameters:

Variable

Required. Variable is any expression for the array to adjust.

NumDimensions

Required. Any numeric expression that gives the number of dimensions for the variable's multidimensional array.

Start

Required. Can be either a numeric expression or an array specifying the start for each of the dimensions.
If Start is a numeric expression, each of the dimensions will start at that value.
If Start is an array, then the Nth element in the array will correspond to the start of the Nth dimension (i.e. dimension 2 will start at start[2]).

Size

Required. Can be either a numeric expression or an array specifying the size of each of the dimensions.
If Size is a numeric expression, each of the dimensions will be the size of that value.
If Size is an array, then the Nth element in the array will correspond to the size of the Nth dimension (i.e. dimension 2 will have a size of Size[2]).

Comments: This statement takes a variable and gives it the array attributes specified by the parameters provided. It can also be used to take an array and make it a simple variable by specifying the number of dimensions as "0". Caution should be taken, as all information in the array will be lost when this statement is executed, even if the result is no change in the array's attributes.

Example:

```
elements[0] = 4;
elements[1] = 5;
...
AdjustArray(OldVariable, 2, 1, elements);
```

In this code example, the variable OldVariable is changed into a two-dimensional array with 4 rows and 5 columns. The elements for both the rows and columns will start numbering at 1 (as per the third parameter). This is the same as declaring:

```
OldVariable[1, 4][1, 5];
```

AdjustCode

Description: Adjusts the offsets and sizes of items stored in the .RUN file within the document file.

Warning: This function may cause irrecoverable alteration of your application. It should be used only by advanced programmers.

Returns: Nothing

Usage: Script Only.

Related to: GetModuleText | GetOneParmText | GetParmText | GetStateText | GetTransitText | GetVariableText | SetModuleText | SetOneParmText | SetParmText | SetStateText | SetTransitText | SetVariableText | TextOffset | TextSize

Function Groups: Compilation and Online Modifications, Advanced Module

Format: AdjustCode(Value, Type, Offset, Size)

Parameters:

Value

Required. The value must be a code value or a Variable. This identifies the module, state, statement, or variable to adjust.

Type

Required. Any numeric expression that explicitly spe-

cifies the VTScada Value Types – Numeric Reference.

Offset

Required. Any numeric expression that specifies the starting point for the code adjustment.

Size

Required. Any numeric expression giving the number of bytes by which to shift all offset values following this entry in the .RUN file.

Comments: This statement enables you to adjust a .RUN file while VTScada is running. If a change is to be made to the .SRC file while the application is running, this code change must also be written into the .RUN file, so that the .RUN and .SRC files will remain synchronized. AdjustCode essentially creates a space into which the applicable information may be inserted; Offset is the point for the insertion; and Size is the size of the inserted block. All code offsets that follow the inserted block are increased by Size to allow for the new location of the old code, and all items that contain the newly inserted block have their size increased by Size. For example, if the newly inserted block was going to have a variable definition written to it, and it was contained inside of a module, the module's size would increase by Size. Note that this function does not actually write any changes into the file; rather, it makes room for the changes to be written by another function.

AlignSelected

Note: Deprecated. Do not use in new code.

Description: Aligns selected graphic objects.

Returns: Nothing

Usage: Script Only.

Function Groups: Graphics

Format: AlignSelected(Object, Mode, Side)

Parameters:

Object

Required. Any object expression giving the module instance where the selected objects are found. They will be aligned as specified by the Mode and Side parameters (defined below).

Mode

Required. Any numeric expression that specifies how the objects will be aligned. The possible values are provided in the following table:

Mode	Alignment Type
0	Align
1	Stretch
2	Size
3	Space
4	Tile
5	Equal space between

Side

Required. Any numeric expression that specifies what side will be used when aligning objects. This also specifies the direction for equal or even spacing.

Side	Meaning
0	Left
1	Aligns on vertical center line
2	Right

3	Top
4	Aligns on horizontal center line
5	Bottom

Comments: This statement is used by the GUI tool bar alignment buttons.

Example:

```
AlignSelected(FindVariable("Graphics", Self(), 0, 1)
 { Find the module's object pointer },
              1, 2 { Stretch the right side to align it  });
```

Related Functions:

FindVariable | Self

AlternateIdCheck

Security Manager Module

Description: Searches the accounts for an account whose AltID matches the parameter value.

Returns: String

Usage: Script Only.

Related to: AlternateLogoff | AlternateLogon | Authenticate | LogOff | QuietLogon | UserCredChange | UserLogonDialog

Format: \SecurityManager\AlternateIdCheck(AltID [, ReturnName]);

Parameters:

AltID

The alternate ID to search for in the user accounts.

ReturnName

Optional. A Boolean. If TRUE, returns the account name. If FALSE or Invalid, returns the account ID.

Comments: Returns either the account name, account ID or Invalid if no matching alternate ID was found.

AlternateIDCheck does not log the user in – it only verifies whether the ID is recognized.

AlternateLogoff

Security Manager Module

Description Synonym for LogOff().

Returns Nothing

Usage Script Only.

Related to: AlternateIdCheck | AlternateLogon | Authenticate | LogOff | QuietLogon | UserCredChange | UserLogonDialog

Format \SecurityManager\AlternateLogoff()

Parameters None

Comments None

AlternateLogon

Security Manager Module

Description Either creates, or attempts to log in using an alternate ID value. See comments.

Returns Boolean

Usage Script Only.

Related to: AlternateIdCheck | AlternateLogoff | Authenticate | LogOff | QuietLogon | UserCredChange | UserLogonDialog

Format \SecurityManager\AlternateLogon(AltID, Logon)

Parameters

AltID

The plaintext alternate ID use.

Logon

Flag, TRUE to stop set of AltID

Comments This module call has two modes of operation.

If the SecurityManager user modification dialog is open and Logon is invalid or FALSE, the AltID string is interpreted as the Alternate ID value for the new or existing user account being configured. The intent here is for devices such as swipe card readers to be used to provide the alternate ID.

If the SecurityManager user modification dialog is not open or if Logon is TRUE, attempts to log on the user whose AltID string matches the supplied parameter.

Returns TRUE if the accept/attempt action was successful, otherwise returns FALSE.

AMax

Description: Array maximum. This function returns the maximum value in a sub-range of a numeric array.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Array, Generic Math

Related to: AMin| Array Functions

Format: AMax(ArrayElem, N)

Parameters:

ArrayElem

Required. A numeric array element. The subscript(s) for the array may be any numeric expression, specifying the starting point for the array maximum search.

N

Required. Any numeric expression giving the number of array elements to use starting at the element given by the first parameter.
If N extends past the upper bound of the lowest array dimension, this computation will "wrap-around" and resume at element 0 until N elements have been processed.

Comments: AMax ignores invalid array entries. It only returns invalid if the array subscript is invalid, N is invalid, or if all of the array entries in the specified range are invalid. If processing a multidimensional array, the usual rules apply to decide which dimension should be examined.

Example:

```
highVal = AMax(trendData[0] { Start at element 0 },
           100 { Search through 100 elements });
```

The above example sets highVal to the highest value of trendData elements 0 to 99. Since this function ignores invalid array elements, highVal will be some valid value unless all elements 0 to 99 are invalid.

AMin

Description: Array minimum. This function returns the minimum value in a sub-range of a numeric array.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Array, Generic Math

Related to: AMax| Array Functions

Format: AMin(ArrayElem, N)

Parameters:

ArrayElem

Required A numeric array element. The subscript(s)

for the array may be any numeric expression, specifying the starting point for the array minimum search.

N

Required Any numeric expression giving the number of array elements to use, starting at the element given by the first parameter.
If N extends past the upper bound of the lowest array dimension, this computation will "wrap-around" and resume at element 0 until N elements have been processed.

Comments: AMin ignores invalid array entries. The function only returns invalid if the array subscript is invalid, N is invalid, or if all of the array entries in the specified range are invalid. If processing a multidimensional array, the usual rules apply to decide which dimension should be examined.

Example:

```
low = AMin(trendData[0] { Start at element 0 },
           100 { Search through 100 elements });
```

The example above sets low to the lowest value of trendData elements 0 to 99. Since this function ignores invalid array elements, low will be some valid value unless all elements 0 to 99 are invalid.

And

Description: Returns the bit-wise AND of its two parameters as a 32-bit unsigned integer.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Bitwise Operation

Related to: Or | Not | XOr | Boolean Logic Operators

Format: And(A, B)

Parameters:

A

Required. Any numeric expression for the first parameter, which is truncated to a 32-bit unsigned number.

B

Required. Any numeric expression for the second parameter, which is truncated to a 32 bit unsigned number.

Comments: The parameters and the result can be up to 32 bits long. If either argument is invalid, the result is invalid.

Example:

```
result = And(0b1010, 0b1100);
```

This example sets the variable result to 0b1000.

AppIsRunning

Description: Reports whether the application has been started and the start-up process is complete. (All tags are running, etc.)

Returns: Boolean

Usage: Script only.
May be used in optimized Tag Parameter Expressions.

Function Groups: Configuration Management

Related to: AppIsStarted | AppIsStarting | Start | IsAppEditable | GetAppInstance | GetLoadedAppInstance | GetOEMLayer

Format: \LayerRoot\AppIsRunning()

Parameters: None

Comments: The similar functions, AppIsStarting and AppIsStarted, will return TRUE before this function will. \LayerRoot may be

acquired using one of GetAppInstance, GetLoadedAp-
pInstance or GetOEMLayer.

Examples:

```
IF 1 NextState;
[
  IsVTSApp = Valid(LayerRoot\OEMGUID);
  IfThen(IsVTSApp && LayerRoot\AppIsRunning()),
    ...
  )
]
```

AppIsStarted

Description: Returns TRUE if the application has been started.

Returns: Boolean

Usage: Script only.
May be used in optimized Tag Parameter Expressions.

Function Groups: Configuration Management

Related to: AppIsRunning | AppIsStarting | Start | GetAppInstance | GetLoadedAppInstance | GetOEMLayer

Format: LayerRoot\AppIsStarted()

Parameters: none

Comments: "Started" is a different state than "Running". Use care when deciding whether to use this function or AppIsRunning.
Like Start, this is typically called on another Layer, rather than one's own layer, which is presumably running.
The Layer object can be acquired using GetAp-
pInstance, GetLoadedAppInstance or GetOEMLayer.

Examples:

```
IF 1 NextState;
[
  IfThen( LayerRoot\AppIsStarted()),
    ...
  )
]
```

AppIsStarting

Description: Returns TRUE if the application is in the process of starting.

Returns: Boolean

Usage: Script only.
May be used in optimized Tag Parameter Expressions.

Function Groups: Configuration Management

Related to: AppIsStarted | Start | AppIsRunning | GetAppInstance | GetLoadedAppInstance | GetOEMLayer

Format: LayerRoot\AppIsStarting

Parameters: none

Comments: Checks whether a call to Start() is being processed in the application specified by LayerRoot.
The Layer object can be acquired using GetAp-pInstance, GetLoadedAppInstance or GetOEMLayer.

Examples:

```
IF 1 NextState;
[
  IfThen(LayerRoot\AppIsStarting()),
    ...
  )
]
```

ApplyChangeSetFile

Description: Apply a named ChangeSet to an application layer.

Warning: This function should be used by advanced users only. Irre-vocable alteration of your application may occur

Returns: Object (module reference)

Usage: Script Only.

Function Groups: Configuration Management

Related to: DirectApply | GetAppInstance | GetLoadedAppInstance |

GetOEMLayer

Format: Layer\ApplyChangeSetFile(User, Comment, FileName, pError, SuppressError, RSema, Superior)

Parameters:

User

Required. The account name, to which the changes will be attributed.

Comment

Required. A comment to be applied to the revision.

FileName

Required. The full path to the ChangeSet file.

pError

Required. A pointer to a value in which the status of the operation will be stored as a numeric value. Defined error codes are as follows:

Code	Meaning
0	Success / No errors
1	Unsupported file version
2	Checksum error
3	File truncated

SuppressError

Optional Boolean. Set to TRUE if the error dialogs are to be suppressed. Defaults to FALSE.

RSema

Optional repository semaphore.

Superior

Optional Boolean. Set to TRUE if the ChangeSet's changes should take precedence in the event of a conflict.

Comments: The act of applying a ChangeSet file may produce unexpected results. Use with caution.
The Layer object can be acquired using GetAppInstance, GetLoadedAppInstance or GetOEMLayer.

Examples:
none.

Arc

Note: Deprecated. Do not use in new code.

Description: Draws an arc on the screen.

Returns: Nothing

Usage: Steady State only.

Function Groups: Graphics

Related to: Circle | GUIArc | GUIPie

Format: Arc(X, Y, Radius, Angle1, Angle2, Width, Color)

Parameters:

X

Required. Any numeric expression giving the X coordinate of the center of the arc on the screen.

Y

Required. Any numeric expression giving the Y coordinate of the center of the arc on the screen.

Radius

Required. Any numeric expression giving the radius of the arc specified in units of X screen coordinates.

Angle1

Required. Any numeric expression giving the starting angle of the arc specified in radians. An angle of 0 lies on the X axis to the right of the center of the arc.

Angle2

Required. Any numeric expression giving the ending angle of the arc specified in radians.

Width

Required. Any numeric expression giving the line width for the arc. The width is specified in terms of X screen coordinates. Any width less than 1 screen pixel is treated as a 1 pixel arc.

Color

Required. Any numeric expression giving one of the reserved VTScada Color Palette values for the arc.

Comments: The Arc statement has been superseded by the GUIArc function and is maintained for backwards compatibility only.
The arc is drawn in a counterclockwise direction from the Angle1 to Angle2.
As of version 11, this is now drawn in the same z-order as other graphics, making it similar to the z-graphics functions.

Example:

```
Arc(500, 500 { X and Y coordinates },
    100 { Radius in screen coordinates },
    0 { Starting angle in radians - 3 o'clock },
    4.71 { Ending angle in radians - 6 o'clock },
    10 { Arc line thickness in screen coordinates },
    15 { Color is white });
```

ArrayDimensions

Description: Returns the number of dimensions in an array.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Array

Related to: ArraySize | ArrayStart| Array Functions

Format: ArrayDimensions(Array)

Parameters:

Array

Required Any array variable.

Comments: The ArrayDimensions function is useful for writing intelligent parametrized modules. The module can determine how many dimensions there are in an array that is passed to it.

Note that ArrayDimensions will not drill down into nested array structures; that is to say, if a 3 dimensional array is created, and a 2 dimensional array is stored in each of its elements, ArrayDimensions will return a value of 3, not 5. This is because the data stored in different array elements does not need to be of the same type – one element could contain an array while another element could contain a simple value.

Note: An Invalid will be returned if a simple value is handed to ArrayDimensions().

Example:

```
x[10][3];
...
y = New(3, 0, 10);
numDimX = ArrayDimensions(x);
numDimY = ArrayDimensions(y);
In this example, NumDimX will have a value of 2, while NumDimY will
have a value of 3.
```

ArrayOp1

Description: Performs a mathematical operation on an array with respect to a scalar value.

Returns: Nothing

Usage: Script only.

May be used in optimized Tag Parameter Expressions.

Function Groups: Array

Related to: ArrayOp2 | ArraySize | FFT| Array Functions| WhileLoop

Format: ArrayOp1(ArrayElem, N, Scalar, OpCode[, MaskElement])

Parameters:

ArrayElem

Required. Any array element giving the starting point for the array operation. The subscript for the array may be any numeric expression. If processing a multidimensional array, the usual rules apply to decide which dimension should be used.

N

Required. Any numeric expression giving the number of array elements to compute. N will be limited to the minimum of (N, Source length, Destination length). If the starting point given by the first parameter is not the first element, and therefore N extends past the upper bound of the lowest array dimension, this computation will "wrap-around" and resume at the first element. Since N is automatically limited by the smallest array dimension, no element will be processed twice.

Scalar

Required. Any numeric expression giving the scalar quantity used in the array computation.

OpCode

Required. Any numeric expression giving the operation number to perform as follows (note that A is the array element, and S is Scalar):

OpCode	Operation	OpCode	Operation
0	A = S (S may be invalid)	17	A = XOR(A,S)

1	A = A + S	18	A = Pow(S,A)
2	A = A – S	19	A = Exp(S*A)
3	A = S – A	20	A = S * Log (A)
4	A = A * S	21	A = S * Ln(A)
5	A = A / S	22	A = Sin(S * A)
6	A = S / A	23	A = Cos(S*A)
7	A = A % S	24	A = Tan (S*A)
8	A = Min(A,S)	25	A = S * ASin (A)
9	A = Max(A,S)	26	A = S * ACos (A)
10	A = A < S	27	A = S * ATan (A)
11	A = A <= S	28	A = S * Sqrt (A)
12	A = A==S	29	A = S * Abs (A)
13	A = A >= S	30	A = (Index of A) + S
14	A = A > S	31	A = (Index of A) * S
15	A = AND(A,S)	32	A = S * Round (A)
16	A = OR(A,S)	33	Valid(A) != Valid(S) ||

PickValid

(A != S, 0)

MaskElement

Optional. The starting element in an array of TRUE and FALSE values, specifying which elements in array, A will be operated upon. The ArrayOp function will only apply to element A[i] if Mask[starting element + i] is a valid TRUE.

Comments: The ArrayOp1 statement is useful for large amounts of repetitive computation. While this can be done by executing a script repeatedly using WhileLoop, this ArrayOp1 statement is much faster. Complex computations may be broken down into a series of simple steps and handled by multiple ArrayOp1 and ArrayOp2 statements. If text arrays are used, each text value will be converted to a number before the numerical operations are performed. The resulting array is converted back to text.

Mode 33 will compare for differences. It does a more thorough comparison than a simple A != B.

Example:

```
If 1 Main;
[
 ArrayOp1(pumpTrend[10] { Starting element },
          50 { Number of elements to use },
          x + y { Number to use in computation },
          0 { Assign result to the element });
 ArrayOp1(sequentNums[0] { Starting element (beginning) },
          10 { Number of elements to use },
          1 { Number to use in computation },
          30 { Assign sequential numbers });
 ArrayOp1(evenNums[0] { Starting element (beginning) },
          101 { Number of elements to use },
          2 { Number to use in computation },
          31 { Assign multiples of number });
]
```

In this example, the first statement sets elements 10 through 59 of pumpTrend to the result of x + y, the second statement sets elements 0 through 9 of sequentNums to the numbers 1 through 10, and the third statement sets elements 0 through 100 of evenNums to the even numbers from 0 to 200.

ArrayOp2

Description: Performs a mathematical operation on an array with respect to another array.

Returns: Nothing

Usage: Script only.
May be used in optimized Tag Parameter Expressions.

Function Groups: Array

Related to: ArrayOp1 | ArraySize | FFT| Array Functions| WhileLoop

Format: ArrayOp2(Array1Element, Array2Element, N, OpCode[, MaskElement])

Parameters:

Array1Element

Required. Any array element giving the starting point for the array operation in the destination array.
The subscript for the array may be any numeric expression. If processing a multidimensional array, the usual rules apply to decide which dimension should be used.

Array2Element

Required. Any array element giving the starting point for the array operation in the source array.
The subscript for the array may be any numeric expression.

N

Required. Any numeric expression giving the number of array elements to compute. N will be limited to the

minimum of (N, Source length, Destination length). If the starting point given by the first parameter is not the first element, and therefore N extends past the upper bound of the lowest array dimension, this computation will "wrap-around" and resume at the first element. Since N is automatically limited by the smallest array dimension, no element will be processed twice.

OpCode

Required. Any numeric expression giving the operation number to perform as follows (note that A is an element of Array1 and B is an element of Array2):

OpCode	Operation	OpCode	Operation
0	A = B	17	A = XOR(A,B)
1	A = A + B	18	A = Pow(A,B)
2	A = A - B	19	A = Exp(B*A)
3	A = B - A	20	A = B * Log(A)
4	A = A * B	21	A = B * Ln(A)
5	A = A / B	22	A = Sin(B * A)
6	A = B / A	23	A = Cos(B * A)
7	A = A % B	24	A = Tan (B * A)
8	A = Min (A,B)	25	A = B * ASin (A)
9	A = Max (A,B)	26	A = B * ACos (A)
10	A = A < B	27	A = B * ATan (A)
11	A = A <= B	28	A = B * Sqrt(A)

12	A = A==B	29	A = B * Abs(A)
13	A = A >= B	30	A = (Index of A) + B
14	A = A > B	31	A = (Index of A) * B
15	A = AND (A,B)	32	A = B * Round (A)
16	A = OR(A,B)	33	Valid(A) != Valid(B) || PickValid(A != B, 0)

MaskElement

Optional. The starting element in an array of TRUE and FALSE values, specifying which elements in array, A will be operated upon. The ArrayOp function will only apply to element A[i] if Mask[starting element + i] is a valid TRUE.

Comments:

The ArrayOp2 statement is useful for large amounts of repetitive computation. While this can be done by executing a script repeatedly using WhileLoop, the ArrayOp2 statement is much faster. Complex computations may be broken down into a series of simple steps and handled by multiple ArrayOp1 and ArrayOp2 statements. If text arrays are used, each text value will be converted to a number before the numerical operations are performed. The resulting array is converted back to text.
Mode 33 will compare for differences. It does a more thorough comparison than a simple A != B.

Example:

```
If 1 Main;
[
 ArrayOp2(xArray[3] { Starting element in xArray },
```

```
          yArray[3] { Starting element in yArray },
          50 { Perform operation on 50 elements },
          4 { xArray element x yArray element });
]
```

The example above sets each element in xArray to the product of that element with a corresponding element in yArray, beginning at element 3 and ending at element 52. In other words:

```
xArray[n] = xArray[n] * yArray[n]
```

for n = 3 to 52.

ArraySize

Description: Returns the number of elements in an array dimension.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Array

Related to: ArrayDimensions | ArrayStart | Array Functions

Format: ArraySize(Array[, Dimension])

Parameters:

Array

Required. Any array variable.

Dimension

Optional. Any numeric expression that gives the array dimension to measure starting at 0 (the left-most dimension in a multi-dimensional array). Defaults to zero.

Comments: If Array is a variable rather than an array, return value will be invalid.
This function is useful for writing intelligent parametrized modules. The module can determine how many elements there are in an array that is passed to it.

Example:

```
q[10];
...
size = ArraySize(Q);
```

In this example, Size will receive the value 10.

ArrayStart

Description: Returns the first element in an array dimension

Returns: Numeric

Usage: Script or steady state.

Related to: ArrayDimensions | ArraySize| Array Functions

Function Groups: Array

Format: ArrayStart(Array, Dimension)

Parameters:

Array

Required. Any array variable.

Dimension

Required. Any numeric expression that gives the array dimension to measure starting at 0.

Comments: The ArrayStart function is useful for writing intelligent parametrized modules. The module can determine the first index in an array that is passed to it.

Example:

```
q[1, 10][2, 20];
...
start = ArrayStart(Q, 1);
```

Start will receive the value 2.

ArrayToBuff

Description: Returns a buffer containing the numeric data from an array.

Returns: Numeric Buffer

Usage: Script or steady state.

Related to: ArrayStart | ArraySize | BuffRead | BuffToArray | BuffToParm | BuffToPointer | BuffWrite | GetByte | MakeBuff | ParmToBuff | PointerToBuff | SetByte| Array Functions

Function Groups: Array, String and Buffer.

Format: ArrayToBuff(ArrayElem, N, Option, Size, Skip [,BadData])

Parameters:

ArrayElem

Required. Any array element giving the starting point for the array conversion. The subscript for the array may be any numeric expression. If processing a multidimensional array, the usual rules apply to decide which dimension should be used.
Note: The array must contain numeric data only.

N

Required. Any numeric expression giving the number of array elements to convert starting at the element given by the first parameter. If N extends past the upper bound of the lowest array dimension, this computation will "wrap-around" and resume at element 0, until N elements have been processed.

Option

Required. Any numeric expression that specifies the format of the buffer write, using the following table of formats:
Note: For Options 7 and 9, the data is written as appropriate binary format.

Option	Buffer Format

0	Unsigned binary (low byte first)
1	Signed binary (low byte first)
2	BCD (binary coded decimal – low byte first)
3	ASCII octal (high byte first)
4	ASCII decimal (high byte first)
5	ASCII hex (high byte first)
6	ASCII floating point (high byte first)
7	IEEE float/double (low byte first)
8	<obsolete>
9	Allen-Bradley PLC/3 floating point
10	VAX single precision floating point

Size

Required. Any numeric expression giving the number of digits in each datum; it has a different meaning for each option as follows:

Size	Size Meaning	Size Range
Binary types	Number of bits	1 – 32 bits
BCD	Number of 4-bit digits	1 – 8 digits
ASCII types	Number of bytes	1 – 32 bytes
Float types	Precision	1 for single precision, 2 for double precision.

Skip

Required. Any numeric expression giving the number of buffer bits/digits/bytes to skip after writing each non-floating point element. For floating point types, this parameter must be set to 0.

BadData

Optional. A parameter that designates how invalid data is to be handled, according to the following table. Defaults to 0 if missing or invalid.

BadData	Invalid Data Type
0	Output to buffer as invalid values
1	Causes buffer to be invalid
2	Output to buffer as valid 0s

Comments: This function may only be used with arrays containing numeric data. It is useful for writing I/O drivers and saving arrays of data in RAM with a fraction of the memory requirement.

Example:

In the following example, assume that array x is a one-dimensional array containing the values 4.5, invalid, and 200:

```
If ! Valid(buff);
[
  buff = ArrayToBuff(x[0] { Starting element },
                     3 { Number of elements to process  },
                     7  { Type - IEEE floating point },
                     1  { Single precision },
                     0  { Skip is ignored },
                     2  { Use 0 for each invalid value });
  BuffRead(buff   { Buffer to read },
           0 { Starting offset },
           "%3b%3b%3b" { Type IEEE floating point binary },
           a1, a2, a3 { Variables to hold the values });
]
```

This code produces a formatted buffer called buff that holds 3 float values (written in binary format), each corresponding to an element in the array. The second value in the array is invalid - in the buffer, it will be a valid 0. The values of a1, a2 and a3 then will be 4.5, 0 and 200 respectively.

ASin

Description: Returns the trigonometric arc sine in radians.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Trigonometric Math

Related to: ACos | ATan | Cos | Sin | Tan

Format: ASin(X)

Parameters:

X

Required. Any numeric expression in the range -1 to +1.

Comments: The returned angle is in radians. To convert an angle from radians to degrees, divide by \pi/ 180 or (approximately) 0.0174533.

Example:

```
radAngle = ASin(1);
degAngle = radAngle / \pi / 180;
```

The value of degAngle will be 90.

ATan

Description: Returns the trigonometric arc tangent in radians.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Trigonometric Math

Related to: ACos | ASin | Cos | Sin | Tan

Format: ATan(X)

Parameters:

X

Required. Any numeric expression.

Comments: The returned angle is in radians. To convert an angle from radians to degrees, divide by \pi / 180 or (approximately) 0.0174533.

Example:

```
radAngle = ATan(1);
degAngle = radAngle / \pi / 180;
```

The value of degAngle will be 45.

AudioFileLength

(System Library)

Description: Returns the length of a RIFF format Wave file in seconds.

Returns: Numeric

Usage: Script or steady state.

Related to:

Function Groups: Speech and Sound

Format: \System\AudioFileLength(Filename)

Parameters:

Filename

Required. Any expression for the full name of a .wav file for which you want to find the audio play length.

Comments: This module is a member of the System Library, and must therefore be prefaced by \System\, as shown in the

"Format" section. If you are developing a script application, use "System\..." rather than "\System\..." in the function call.
Returns invalid if the file does not exist, or if the length cannot be determined.

Example:

```
PlayingTime = AudioFileLengh(AlarmMessage.wav);
```

Authenticate

Security Manager Module

Description Authenticates the Namespace, UserName and Password.

Returns Boolean

Usage Script Only.

Function Groups: Security

Related to: AlternateIdCheck | AlternateLogoff | AlternateLogon | LogOff | QuietLogon | UserCredChange | UserLogonDialog

Format \SecurityManager\Authenticate(UserName, PassWord [,Namespace, Privilege, Device, DontLog]);

Parameters

UserName

The username to authenticate with.

Password

The password to validate against the given UserName.

Namespace

Optional. The namespace of the user. Defaults to none.

Privilege

Optional. A privilege that the account must pass a SecurityCheck with. Defaults to none.

Device

Optional. Name of the device that is making the request. Defaults to none. Only used for security log messages.

DontLog

Optional. A Boolean. If TRUE, the result of the Authenticate request will not be logged in the security event log. Defaults to FALSE.

PtrAccountID

Optional. A pointer to a variable that the AccountID of the user will be stored in.

Comments If the authentication fails, a failure event is recorded in the security event log and FALSE is returned.
If the authentication succeeds, TRUE is returned and the AccountID of the user is written into the variable addressed by the PtrAccountID parameter.
If Privilege is valid, the Privilege is used in a SecurityCheck and the result of the SecurityCheck is returned.

AValid

Description: Returns the number of valid elements in an array sub-range.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Array.

Related to: AMax | AMin | ArrayOp1 | ArrayOp2 | FiltHigh | FiltLow | FitOffset | FitSlope | Mean | SDev | Sum| Valid | Variance |

Threaded: No.

Format: AValid(ArrayElem, N)

Parameters:

ArrayElem

Required. An array element. The subscript(s) for the array may be any numeric expression and specify the starting point for the array search.
If processing a multidimensional array, the usual rules apply to decide which dimension should be used.

N

Required. Any positive numeric expression giving the number of array elements to use, starting at the element given by the first parameter. If N extends past the upper bound of the lowest array dimension, this computation will "wrap-around" and resume at element 0, until N elements have been processed.

Comments: AValid is not made invalid by invalid array entries. This function is useful in conjunction with the statistical array functions.

Example:

```
numValid = AValid(data[0] { Starting element },
                  100 { Number of elements to search });
```

The example above finds the number of array elements with valid values, by examining elements 0 to 99.

B Functions

The sections that follow identify all VTScada functions beginning with "B".

Ball

Note: Deprecated. Do not use in new code.

Description: Draws a filled circle on the screen.

Returns: Nothing

Usage: Steady State only.

Function Groups: Graphics

Related to: Circle | Ellipse | GUIEllipse

Format: Ball(X, Y, Radius, Foreground, Pattern, Background)

Parameters::

X

Required. Any numeric expression giving the X coordinate of the center of the ball on the screen.

Y

Required. Any numeric expression giving the Y coordinate of the center of the ball on the screen.

Radius

Required. Any numeric expression giving the radius of the ball specified in units of X screen coordinates.

Foreground

Required. Any numeric expression giving the foreground VTScada Color Palette of the ball.

Pattern

Required. Any numeric expression giving the hatch pattern to use to fill the ball. The valid hatch style numbers are from 1 to 25 inclusive. A Pattern of 1 is a solid ball (for other patterns, please refer to Fill Patterns).

Background

Required. Any numeric expression giving the background VTScada Color Palette for the hatch pattern used to fill the ball.
This value is only significant if the Pattern parameter is not equal to 1 (solid).

Comments:

This statement has been superseded by the GUIEllipse function and is maintained for backwards compatibility only.
As of version 11, this is now drawn in the same z-order as other graphics, making it similar to the z-graphics functions.

Example:

```
Ball(20, 80 { Center },
     50 { Radius },
     11 { Cyan foreground },
     21 { Checkered pattern },
     0 { Black background });
```

This draws a cyan and black checkered ball in the upper left portion of the screen.

Bar

Note: Deprecated. Do not use in new code.

Description: Draws a filled bar on the screen.

Returns: Nothing

Usage: Steady State only.

Function Groups: Graphics

Related to: Box | GUIRectangle | ZBox | Scale

Format: Bar(X1, Y1, X2, Y2, Foreground, Pattern, Background)

Parameters:

X1

Required. Any numeric expression giving the X coordinate one side of the bar on the screen (either left or right).

Y1

Required. Any numeric expression giving the Y coordinate of either the top or bottom of the bar on the screen.

X2

Required. Any numeric expression giving the X coordinate of the side of the bar opposite to X1 on the screen (either left or right).

Y2

Required. Any numeric expression giving the Y coordinate of either the top or bottom of the bar, whichever is the opposite to Y1.

Foreground

Required. Any numeric expression giving the foreground VTScada Color Palette of the bar.

Pattern

Required. Any numeric expression giving the hatch pattern to use to fill the bar. The valid hatch style numbers are from 1 to 25 inclusive. A Pattern of 1 is a solid bar. For valid pattern numbers, please refer to Fill Patterns).

Background

Required. Any numeric expression giving the background color for the hatch pattern used to fill the bar. This value is only significant if the Pattern parameter is not equal to 1 (solid).

Comments

This statement has been superseded by the GUIRectangle and ZBar statements, and is maintained for backwards compatibility only.
As of version 11, this is now drawn in the same z-order as other graphics, making it similar to the z-graphics functions.

Note: It is recommended that you use bars without hatch patterns for animation, as solid bars are drawn faster.

Example:

```
Bar(700, 30, 740, 60 { Coordinates outlining the bar },
    2 { Dark green foreground },
```

```
    4 { Striped hatch pattern },
    10 { Bright green background });
```

This example draws a small bar in the upper right corner of the screen using a two-tone green striped hatch pattern.

```
Bar(10, 600 { X and Y coordinates of lower left corner },
    200 Scale(temperature, 0, 150, 100, 500)
    { Make upper right corner move with temp },
    4, 1, 0 { Solid red color });
```

The example above displays a vertical bar graph of temperature. Note that the Y2 coordinate is dependent on a scale of temperature. Every time the value of temperature changes, the Bar will redraw itself to the new upper right corner. The Scale function transforms the temperature (0 to 150) to screen coordinates (100 to 500). However, if temperature should go below 0 or above 150, the bar will still be drawn to the corresponding screen coordinates (above 100 or below 500). To limit the action of the moving bar, try the following:

```
Bar(10, 600 { X and Y coordinates of lower left corner},
    200, Scale(Limit(temperature, 0, 150), 0, 150, 100, 500)
    { Make upper right corner move with temperature},
    4, 1, 0 { Solid red color });
```

This limits the value of temperature used in the scale (it does not limit the value of the temperature variable itself).

Base64Decode

(System Library)

Description: Performs a Base64 decode of a buffer.

Returns: Buffer

Usage: Script Only.

Function Groups: Encryption

Related to: Base64Encode | Hash

Format: \System\Base64Decode(Buffer[, Offset, isMIME]);

Parameters:

Buffer

Required. The encoded value.

Offset

Optional numeric. Offset into the buffer, at which to start decoding. Defaults to zero if invalid.

isMIME

Optional Boolean. If TRUE, relaxes the constraint regarding characters that aren't in the base64 alphabet, per RFC2045. Defaults to FALSE if invalid.

Comments: This function complies with IETF RFC4648 and RFC2045 .
Returns Invalid if the input buffer is invalid, or if IsMime is TRUE and the buffer contains invalid characters as per RFC2045.

Examples:
none

Base64Encode

(System Library)

Description: Performs a Base64 encode of a buffer.

Returns: Buffer

Usage: Script Only.

Function Groups: Encryption

Related to: Base64Decode | Hash

Format: \System\Base64Encode(Buffer);

Parameters:

Buffer

Required. The information to be encoded.

Comments: none.

Examples:

```
EncodedResult = \System\Base64Encode(SomeBuffer);
```

Beep

Description: Causes a tone to sound on the computer's internal speaker.

Returns: Nothing

Usage Script or steady state.

Function Groups: Speech and Sound

Related to: Sound

Format: Beep(Frequency)

Parameters::

Frequency

Required. Any numeric expression giving the frequency to be output to the speaker.

Comments:

The minimum frequency is 1190000 / 65535 ~= 18.16. If the frequency is set below the minimum frequency or invalid, the speaker will turn off.

Note: Within an Anywhere Client session, this function does nothing.

Example:

```
If MatchKeys(2, "on");
[
  Beep(2000) { A 2kHz tone };
]
If MatchKeys(2, "off");
[
  Beep(0) { No tone };
]
```

The example above results in a 2000 Hz sustained beep sounding when the user types "on" until the user types "off".

Bevel

(System Library)

Description: Draws a titled beveled box.

Returns: Nothing

Usage Steady State only.

Function Groups: Graphics

Related to: TextBox | Edit

Format: \System\Bevel(X1,Y1, X2, Y2 [,Title, AlignTitle, Color])

Parameters:

X1

Required. Any numeric expression giving the X coordinate on the screen of one side of the bevel. The smaller of X1 or X2 will always be the left side.

Y1

Required. Any numeric expression giving the Y coordinate on the screen of either the top or bottom of the bevel. The smaller of Y1 or Y2 will always be the top edge.

X2

Required. Any numeric expression giving the X coordinate on the screen of the side of the bevel opposite to X1.

Y2

Any numeric expression giving the Y coordinate on the screen of the top or bottom of the bevel, whichever is the opposite to Y1.

Title

Optional. Any text expression to be used as a title embedded in the bevel. The default value is a null text string.

AlignTitle

Optional. Any logical expression. If true (non-0) the

top of the title will be aligned with the top of the defined area, if false (0) the bevel will be aligned with the top of the area and the title will protrude past the top.
The default value is true.

Color

Optional. Any numeric expression giving the index of the VTScada Color Palette to be displayed under the title.
Note that this will affect the area immediately under the title (i.e. it does not affect the area inside the bevel). No default value is provided.

Comments:

Bevel is a member of the System Library, and must therefore be prefaced by "\System\", as shown in the "Format" section above. If you are developing a script application, use "System\..." rather than "\System\..." in the function call.
For any optional parameter that is to be set, all optional parameters preceding the desired one must be present, although they may be invalid.

Example:

```
\System\Bevel(10, 10, 110, 40, "Description");
```

BinIP2Text

(RPC Manager Library)

Description: Returns a text representation of a specified binary IP in a printable format.

Returns: Text

Usage: Script Only.

Function Groups: Network, String and Buffer

Related to: TextIP2Bin

Format: \RPCManager\BinIP2Text(BinIP)

Parameters:

BinIP

Required. The Binary representation of the IP to be converted to a printable format.

Comments:

This subroutine is a member of the RPC Manager's Library, and must therefore be prefaced by \RPCManager\, as shown in the "Format" section. If the application you are developing is a script application, the subroutine call must be prefaced by System\RPCManager\, and the System variable must be declared in AppRoot.src.
Subroutine call.

Bit

Description: Returns the on/off status of a bit in a number.

Returns: Boolean

Usage Script or steady state.

Function Groups: Bitwise Operation

Related to: And | SetBit

Format: Bit(*Value, BitNumber*)

Parameters:

Value

Required. Any numeric expression giving the number containing the bit to be tested.

BitNumber

Required. Any numeric expression in the range of 0 to 31 giving the bit number to be tested within the number specified by the Value parameter.
Bit 0 is the least significant bit. Any value outside the range of 0 to 31 will result in a false res-

ult for Bit.

Comments:

Bit returns true (1) if the indicated bit is 1, and false (0) if the indicated bit is 0.

Example:

```
motorFault = Bit(motorStatus, 0);
```

The example above sets motorFault to the value of the least significant bit (LSB) of motorStatus.

BitmapInfo

Description: Returns information about an image.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Graphics

Related to: Crop | MakeBitmap

Format: BitmapInfo(*BitmapVal*, *Attribute*)

Parameters:

BitmapVal

Required. Any expression that returns an image value such as MakeBitmap()

Attribute

Required. Any numeric expression that identifies the information to be returned by this function, as per the following table:

Attribute	Information returned
0	Bitmap width in pixels
1	Bitmap height in pixels

Comments:

BitmapInfo can be used to automate a module that displays images in response to a change in the image.

Example:

```
pumpWidth = BitmapInfo(pumpBitmap, 0);
```

The example displayed above sets pumpWidth to the width of the image value pumpBitmap.

Blend

System layer

Description: Returns an aRGB color value that is a given percentage between two specified colors.

Returns: aRGB color string

Usage: Steady State only.

Function Groups: Color

Related to:

Format: \System\ColorTools\Blend(Color1, Color2, Ratio)

Parameters:

Color1

Required. Any text expression giving the aRGB value of the first color.

Color2

Required. Any text expression giving the aRGB value of the second color.

Ratio

Optional. Any numeric expression for a value between 0 and 1. The larger the Ratio, the closer the resulting color will be to Color1. Default = 0.5

Comments: Returns an aRGB value for a color that is a blend of Color1 and Color2. By default, the values will be

averaged, but any ratio between the two colors can be specified. Blend is especially useful for creating accent colors in dialog boxes.

Example:

```
LineColor = PickValid(\System\ColorTools\Blend(TextColor, BGColor,
0.1), TextColor);
```

Sets the color of an accent line to be similar to a preset background color but with 10% of the preset text color.

Related Information:

BlockDecrypt

(System Library)

Description: Decrypts a value that was encrypted using the Block-Encrypt function.

Returns: String

Usage: Script Only.

Function Groups: Encryption

Related to: BlockEncrypt | Decrypt | Hash

Format: \System\BlockDecrypt(CipherValue, Key);

Parameters:

CipherValue

Required. The string that was encrypted using the BlockEncrypt function

Key

Required string. The key value that was used for encryption.

Comments:

Examples:

```
DecodedResult = \System\BlockDecrypt(SomeEncryptedValue, SameKey);
```

BlockEncrypt

(System Library)

Description: Uses encryption to encode a VTScada value into a Base64 string.

Returns: String

Usage: Script Only.

Function Groups: Encryption

Related to: BlockDecrypt | Encode | Base64Encode | Hash

Format: \System\BlockEncrypt(PlainValue, Key);

Parameters:

PlainValue

Required. The information to be encoded.

Key

Required string. The key value to be used for encryption.

Comments: The resulting string will be different each time it is encoded, even if the same key is used.

Examples:

```
EncodedResult = \System\BlockEncrypt(SomeValue, SomeKey);
```

BlockWrite

Description: Writes a block of data to a stream.

Returns: Nothing

Usage: Script Only.

Function Groups: Stream and Socket

Related to: FWrite | PipeStream | SWrite

Format: BlockWrite(Stream, Data [, ByteLimit])

Parameters:

Stream

Required. Any expression that returns a stream value.

Data

Required. Any text or stream expression that specifies the block of data to write to Stream.

ByteLimit

Optional Any numeric expression giving the maximum number of bytes copied from a stream.

Comments:

BlockWrite is more efficient than writing a block of data character-by-character. It is especially useful for named pipe streams when the intent is to send a block of bytes as a single message, rather than as a series of single-character messages (which is the result of using SWrite with the "%s" format option). If the second parameter is a stream value, the entire stream will be written to the first stream value.

Example:

```
if 1 NextState;
[
 stream = BuffStream("") { Open the first stream };
 SWrite(stream { Write to first stream },
        "%s %d %s" { Format for the data },
        "Test number: ", testNum, "is complete" { The data to
write });
 file = FileStream("TestStatus" { Open second stream });
 Seek(stream, 0, 0) { Rewind the stream });
 BlockWrite(file, stream { Write from stream to file });
 CloseStream(stream);
]
```

The BlockWrite function can be used to copy files as shown in the following example:

```
BlockWrite(FileStream ("File2.ext"), FileStream ("File1.ext"));
```

When used in a manner similar to the example shown above, the BlockWrite statement is particularly useful in copying files (the DOS equivalent

of "copy file1.ext file2.ext"), except in cases where File2 is larger than File1. In cases where File2 is larger than File1, the result (File2) will be its original size. All the data from File1 will overwrite the beginning portion of the data in File2, with the remainder of the data originally contained in File2 remaining untouched.
If you require an exact copy of File1 (rather than the File1 data appended with the remaining File2 data), you can use an FWrite command to delete the destination file before calling BlockWrite.

Boolean

(System Library)

Description: Takes a valid Boolean test and returns the numeric equivalent.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Logic Control

Related to: Boolean Logic Operators | Case | Cond | IF | IfElse| IfOne | IfThen | PickValid

Format: \System\Boolean(Variable)

Parameters:

Variable

Required. The variable whose value is to be converted.

Comments:

This module is a member of the System Library, and must therefore be prefaced by \System\, as shown in the "Format" section. If you are developing a script application, use "System\..." rather than "\System\..." in the function call.
The return value from this module is 1 if Variable contains a valid Boolean true of any case ("true", "t", "yes", "y", "on") and 0 otherwise.

Example:

```
Var = PickValid(System\Boolean(Var), 0);
```

Box

Note: Deprecated. Do not use in new code.

Description: Draws an empty box on the screen.

Returns: Nothing

Usage: Steady State only.

Function Groups: Graphics

Related to: Box | GUIRectangle | ZBox | ZBar

Format: Box(X1, Y1, X2, Y2, Style, Width, Color)

Parameters:

X

Required. Any numeric expression giving the X coordinate of one side of the box on the screen. The smaller of X1 and X2 will always be the left side.

Y1

Required. Any numeric expression giving the Y coordinate of either the top or bottom of the box on the screen. The smaller of Y1 and Y2 will always be the top edge.

X2

Required. Any numeric expression giving the X coordinate of the side of the box opposite to X1 on the screen (either left or right)..

Y2

Required. Any numeric expression giving the Y coordinate of either the top or bottom of the box, whichever is the opposite to Y1.

Style

Required. Any numeric expression giving the line style for the box wall. Valid line styles are from 1 to 10

inclusive. A line style of 1 is a solid line.

Width

Required. Any numeric expression giving the width of the box wall in units of X screen coordinates. The width is always rounded to result in an odd number of pixels on the screen. The minimum width displayed will be 1 pixel.

Color

Required. Any numeric expression giving the VTScada Color Palette of the box.

Comments:

Box has been superseded by the GUIRectangle and ZBox statements, and is maintained for backwards compatibility only.
As of version 11, this is now drawn in the same z-order as other graphics, making it similar to the z-graphics functions.

Example:

```
Box(700, 500, 780, 580 { Bounding box },
    1, 0 { Solid line style, one pixel wide },
    9 { Bright blue color });
```

The example above draws a bright blue box in the lower right-hand corner of the screen.

Brush

Description: Returns a brush value.

Returns: Numeric

Usage: Steady State only.

Function Groups: Color, Graphics

Related to: Pen | GUIPie | GUIEllipse

Format: Brush(Foreground, Background, Pattern)

Parameters:

Foreground

Required. Any numeric expression giving the foreground color of the brush pattern. Any of the following may be used:

- a palette index color
- a system color
- -1 (transparent)
- an RGB string in the format, "<RRGGBB>"

Background

Required. Any numeric expression giving the background color of the brush pattern. If there is a solid pattern, this parameter is ignored.

Any of the following may be used:

- a palette index color
- a system color
- -1 (transparent)
- an RGB string in the format, "<RRGGBB>"

Pattern

Required. Any numeric expression giving the hatch Fill Patterns to use.

A 0 is an invisible pattern, and a 1 is a solid pattern. The maximum style value is 25.

Comments:

Brush values are used in layered graphics statements that paint areas (such as GUIPie or GUIEllipse).

This function may not appear in a script.

Example:

```
newStyle = Brush(12 { Red },
                 14 { Yellow },
                 25 { Brick pattern });
```

The example above creates a brush composed of red bricks outlined in yellow.

BuffOrder

Description: Reverses the order of groups of bytes in a buffer, and returns a new (rearranged) buffer.

Returns: Buffer

Usage: Script or steady state.

Function Groups: String and Buffer

Related to: BuffRead | BuffWrite | Replace | Reverse | ParmToBuff | BuffToParm

Format: BuffOrder(Buffer, Offset, Size, Increment, N)

Parameters:

Buffer

Required. Any text or buffer expression to be reordered.

Offset

Required. Any numeric expression giving the initial position within the buffer for the ordering, starting at 0. Offset must be greater than or equal to 0.

Size

Required. Any numeric expression greater than 0, giving the size of each group of bytes. Bytes within a group will remain in their original order. The portion of the original buffer that will be re-ordered must be evenly divisible into groups of Size.

Increment

Required. Any numeric expression greater than 0, giving the number of groups of Size bytes to be reordered. Re-ordering is done to the Size-groups within Increments, not across Increments.

N

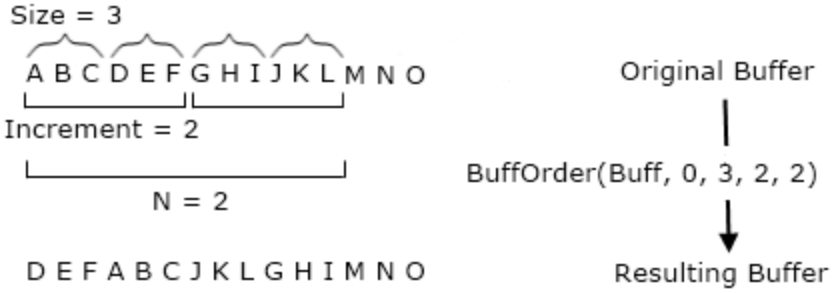
Required. Any numeric expression giving the total

number of Increment-groups to reverse.

Comments:

BuffOrder is part of the driver toolkit. The return value is a buffer containing all bytes specified in the Buffer parameter with groups of bytes reversed. Offset + Size * Increment * N must be no greater than the length of Buffer. Returns Invalid if any parameter does not meet its requirements. BuffOrder is useful for altering buffers for use with the ParmToBuff or BuffToParm functions.

Example:




```
buff1 = BuffOrder("0123456789" { Buffer },
                  0 { Starting offset },
                  1 { Size of group },
                  10 { groups in shuffle },
                  1 { Number of repetitions });
```

In this example, Buff1 will be equal to "9876543210".

```
buff2 = BuffOrder("0123456789" { Buffer },
                  2 { Starting offset },
                  1 { Size of group },
                  2 { Groups in shuffle },
                  4 { Number of repetitions });
```

In this example, Buff2 will be equal to "0132547698".

BuffRead

Description: Reads values from a formatted buffer and returns the number of values not read.

Returns: Numeric

Usage: Script Only.

Function Groups: String and Buffer

Related to: BuffOrder | BuffWrite | FRead | GetByte | SRead | SubStr

Format: BuffRead(Buffer, Offset, Format, V1, V2, V3, ...)

Parameters:

Buffer

Required. Any text or buffer expression to read.

Offset

Required. Any numeric expression giving the initial buffer position for the read, starting at 0.

Format

Required. Any text expression giving the format of how the values (Vn parameters) are to be read.
This format is similar, but not identical, to the C language format string for the scanf function, whereby each of the % format specifications assigns a value to one of the Vn parameters in the statement in the order in which each appears in the list.
Note that like a standard text string, these format specifiers must also be enclosed by double quotes. If a format specification appears for which there are no remaining V parameters, the format specification value is read and discarded.

For the % format specifications, the following form applies (where the [ ] indicates optional elements):
%[*][width]type

Where...

% is mandatory;
The optional asterisk ***** causes the read to occur as per the format specification, but suppresses any assignment to the Vn parameters; and **width** is mandatory, specifying the maximum number of characters to read.
The specifications for **type** are listed in the following table:

Note: Note: Format strings are case insensitive. Additionally, specifying a character for a type that is not in this list results in all the characters following the % up to that point to be read exactly as they appear in the Format string and discarded.

Type	Meaning
Nb	Binary format, where n is a number indicating the type of value (see below)
c	Single ASCII character (byte)
d	Signed decimal integer
e	Signed exponential
f	Signed floating point
g	e or f formats
i	Signed decimal integer
l	Line of characters terminated by a carriage return, line feed, or both
n	Present offset in the buffer
o	Unsigned octal
s	Text string

u	Unsigned decimal integer
x	Unsigned hex integer using "abcdef"
znnn	Escape character where nnn is the 3-digit ASCII code

nb, Binary type For the format specification of %nb, where n specifies the type of number, n must be a single digit from one of the following choices. All are low-byte-first.

n value	Type
0	Byte
1	Short integer (2 bytes, low byte first)
2	Long integer (4 bytes, low bytes first)
3	IEEE single precision float (4 bytes)
4	<obsolete>
5	IEEE double precision float (8 bytes)
6	<obsolete>
7	Binary unsigned short (2 bytes, low byte first)
8	Unsigned 32-bit integer

c, ASCII character type: Unlike BuffWrite this type deals with characters in a string; each character being equal to one byte. Unlike the **%s** option, which reads only up to the first white-space character, the **%c** option reads the number of characters/bytes specified by its width and is not terminated by any particular character. If no width is specified, a single character is read.

d, Signed decimal integer

e, Signed exponential

f, Signed floating point

g, e or f formats

i, Signed decimal integer type: This option normally reads a decimal integer; however, if a leading "0b" is encountered, the number will be interpreted as binary. If a leading "0" (zero only) is encountered, the number will be interpreted as octal. If a leading "0x" is encountered, the number will be interpreted as hexadecimal.

l, Line of characters: This option reads a line of characters terminated by a carriage return, a line feed, or both (in either order). The carriage return and line feed will be discarded, and the next character read will be the first character on the next line. The maximum number of characters read is 4096 (or less if the width option is used).

n, Buffer offset: This option does not read a value, but returns the present offset in Buffer and can be useful in subsequent reads.

o, Unsigned Octal

s, Text string type: Text in the string is read up

until a white-space character is encountered, or the specified width has been read, whichever is smaller. Square brackets enclosing a character, group of characters, or a caret and a group of characters used in the format string reads strings not delimited by spaces. This is a substitute for the %s format specification. The input is read up to the first character that does not appear inside the square brackets (note that this is case-sensitive). A dash may be used to specify a range of characters. For example, the following format specifier:

% [A-Fa-f]

will read a string up to the first which is not an A, B, C, D, E, or F both upper and lower case.

The caret symbol ^. If the first character inside the square brackets is a caret (^), the read progresses up to, but not including, the first character that appears inside the square brackets:

%[^X-Z]

This would read a string up to, but not including, the first X, Y or Z (upper-case only); if the string were terminated by an X, the next character read would be that X. Inside the square brackets, the backslash is used as an escape character - any character following a backslash (such as a caret, dash, or backslash) is taken as that character without special meaning. For example:

%[^X-Z\^]

would behave as described previously, except that the string would now be read up to but not

including the first X, Y, Z, or ^.
Since format specifications for the Vn parameters are indicated by a percentage sign, to read (and discard) an actual percentage sign as part of the text string, precede it with a backslash character (i.e. \%). Also, since the backslash character is used in this manner, as well as with special control characters such as line feed, carriage return and form feed, to read and discard a backslash, use two backslash characters (i.e. \\).

x, Hexadecimal characters: the %x option reads the number of characters/bytes specified by its width and is not terminated by any particular character. If no width is specified, it will continue reading all bytes that can be recognized as hexadecimal characters. For example, given the string "...= 3D", %[^=]=%2x would read the hedadecimal value, 3D (decimal value, 61).

znnn, Escape characters: This specifies an escape character that will be thrown away when read, where nnn is a 3-digit number giving the ASCII character code of the escape character. This option is generally used as the sole format specifier that reads an entire string, spaces included, discarding every single occurrence of an escape character, or the first occurrence of every pair of escape characters. For example, if the string to be read looked like:

abXc dXXfghiXXXjXXXXkl mX Xn o

and the format specifier indicated that the ASCII code for 'X' (88) was to be the escape code:

%25z088

then the variable that this was read into would contain:

abc dXfghiXjXXkl m n o

Notice that for each occurrence of X, the character immediately following it is saved, even if it is itself an escape character. Then the next occurrence of the escape character is discarded, with the character following it being saved, regardless of what it is, and so on. The width field specifies the maximum number of bytes to place in the output string; if this number is smaller than the input string (less the offending escape characters), the string will be truncated. If no width is specified, a single character will be read.

Control characters: In order to encode certain control characters as part of the Format parameter, one of two methods may be used. The first is to use a backslash character followed by one of the single character codes listed below to produce the desired result. Please note that the letters must be lower case.

Code	Meaning

\b	Backspace
\f	Form Feed
\n	Line Feed
\r	Carriage Return
\t	Horizontal Tab
\v	Vertical Tab

In addition to the predefined codes, an alternate form may be used:

\nnn: where nnn is a three digit integer in the range of 0 to 255 specifying a certain ASCII character. If the number contains less than three digits, the leading spaces must be padded with zeroes; this is not the case with the previously listed single character control characters. For example, to include the one byte ASCII character G in the output, you could place its decimal equivalent of 71 in the Format string as \071.

V1, V2, V3

Optional. Parameters specifying the variables to be read in the form described by the Format parameter.
Expressions are not allowed.
Each of the Vn parameters is read in the order in which each appears in the parameter list. V1 has the format given by the first % sequence in the Format parameter, V2 has the second, and so forth.

Comments:

In early versions of VTS (WEB), there was a numeric leading parameter, N. This should not be included in any new code.
The strings read by this function may be part of an I/O driver (such as a message packet), or entered from the keyboard (such as a command-line-interpreter interface). The return value is optional and is the number of Vn parameters not read; this can be used as an error flag.

Example:

```
If ! Valid(wellLevel);
[
 BuffRead(fileData { Source buffer to read from },
          0 { Starting offset, in characters },
          "%f%d/%d/%d" { Format string },
          wellLevel { First variable - read by %f },
          sampleDay { Second variable - read by %d },
          sampleMonth { Third variable - read by %d },
          sampleYear { Fourth variable - read by %d });
]
```

The example shown above reads a floating point variable and 3 integers. The slash characters "/" must be present in the fileData buffer; they will be read and ignored. If the slash characters "/" are missing, reading will be terminated when one is missed; all variables after that point will be set invalid. If fileData contained:

```
"12.3 28/09/92"
```

wellLevel would be set to 12.3, sampleDay to 28, sampleMonth to 9, and sampleYear to 92. If it is desired to skip a single character, use the "%*c" option (read character and discard).

BuffStream

Description:	Returns an in-memory read/write (expanding) buffer stream.
Returns:	Buffer
Usage:	Script Only.
Function Groups:	Stream and Socket, String and Buffer
Related to:	BuffRead | BuffWrite | CloseStream | GetStreamLength |

PeekStream | Seek | ShiftStream | SRead | StreamEnd | SWrite

Format: BuffStream(Buffer)

Parameters:

Buffer

Required. Any text or buffer expression. This serves as the initial content of the stream.

Comments:

This function returns a stream with the contents of Buffer. The pointer at which an action (read or write) begins will be at the start of the stream. Writing to this stream can overwrite or expand (or both,) the size of this initial stream. In-memory stream can be considerably faster to work with than disk-based streams. Streams can be easier to work with than their older text-buffer counterparts.

Example

```
stream = BuffStream("0123456789");
```

In this example, the variable stream would contain "0123456789".

BuffToArray

Description: Reads an array from a formatted buffer containing numerical data and returns the number of elements read.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Array, String and Buffer

Related to: ArrayToBuff | BuffToParm | BuffToPointer

Format: BuffToArray(ArrayElem, N, Buffer, Offset, Option, Size, Skip)

Parameters:

ArrayElem

Required. Any array element giving the starting point in the destination array where the data is to be stored. The subscript for the array may be any numeric expression.

N

Required. Any numeric expression giving the number of array elements to read, starting at the element given by the first parameter. If N extends past the upper bound of the array dimension, this computation will "wrap-around" and resume at element 0, until N elements have been processed.

Buffer

Required. Any text or buffer expression to read. The data stored in the buffer must be numeric.

Offset

Required. Any numeric expression giving the starting buffer position for the read in characters (bytes), starting at 0.

Option

Required. Any numeric expression that specifies the format of the buffer read. This can be one of:

Option	Buffer format
0	Unsigned binary (low byte first)
1	Signed binary (low byte first)
2	BCD (binary coded decimal) (low byte first)
3	ASCII octal (high byte first)
4	ASCII decimal (high byte first)
5	ASCII hex (high byte first)
6	ASCII floating point (high byte first)
7	IEEE float/double (low byte first)
8	<obsolete>
9	Allen-Bradley® PLC/3 floating point
10	VAX single precision floating point

For Options 7 and 9, the data is read as appropriate binary format.

Size

Required. Any numeric expression giving the number of digits in each datum; it has a different meaning for each option, as indicated below.

Size	Size Meaning	Size Range
Binary types	Number of bits	1 – 32 bits
BCD	Number of 4-bit digits	1 – 8 digits

ASCII types	Number of bytes	1 – 32 bytes
Float types	Precision 1 for single precision, 2 for double precision.	

Skip

Required. Any numeric expression giving the number of buffer units to skip after each element is read. Units are bits for Options 0 and 1, BCD digits for type 2, and characters or bytes for all others.

Comments:

BuffToArray may only be used with buffers containing numeric data. Illegal characters embedded in the ASCII string stop further interpretation and are ignored. If an illegal character or the end of a buffer is encountered before a valid number, the remaining array elements are set to invalid. The return value is the number of array elements read.

Example:

```
BuffToArray(data[0] { Starting array element },
            100 { No. of elements to read from buffer },
            response { Text expression to read },
            0 { Starting offset in buffer, in chars },
            2 { Read BCD format },
            4 { 4 BCD characters is 2 bytes },
            0 { Don't skip anything });
```

This example reads 100 BCD format numbers into array elements 0 to 99. Each BCD number has 4 digits, which occupy 2 buffer bytes.

BuffToHex

(System Library)

Description: Convert a buffer of text data to a string of hexadecimal values representing the ASCII code of each character in the buffer.

Returns: Text

Usage: Script Only.

Function Groups: String and Buffer

Related to: WkStaInfo

Format: \System\BuffToHex(BinaryBuffer)

Parameters:

BinaryBuffer

Required. The buffer that is to be converted to a hexadecimal string.

Examples:

```
\System\BuffToHex("ABCD")
```

...will return "41424344"

```
\System\BuffToHex(10)
```

...will return "3031". 10 is interpreted as the string "10", not the numeric, decimal value.

```
\System\BuffToHex(Concat(MakeBuff(1, 0xFF),
                         MakeBuff(1, 0x00),
                         MakeBuff(1, 0xAA)))
```

...will return "FF00AA"

```
MachineIDString = \System\BuffToHex(WkStaInfo(3));
```

...will return a hexadecimal representation of the workstation's unique identifier.

BuffToParm

Description Convert buffer of numeric data to parameters. This function reads module parameters from a formatted buffer containing numerical data and returns the number of data items read.

Returns Numeric

Usage Script Only.

Function Groups String and Buffer, Compilation and On-Line Modifications

Related to: BuffToArray | Parameter | ParmToBuff

Format BuffToParm(Object, Index, Buffer, Offset, N, Option, Size, Skip)

Parameters

Object

Required. The object value of the module containing the destination parameters.

Index

Required. Any numeric expression giving the first parameter to read, starting at 1.

Buffer

Required. Any text or buffer expression to read.

Offset

Required. Any numeric expression giving the starting buffer position for the read in characters (bytes), starting at 0.

N

Required. Any numeric expression giving the number of parameters to read from the buffer. If there are fewer actual parameters than N specifies, this function continues to the last parameter and then stops.

Option

Required. Any numeric expression that specifies the format of the buffer read. This can be one of:

Option	Buffer format
0	Unsigned binary (low byte first)
1	Signed binary (low byte first)
2	BCD (binary coded decimal) (low byte first)
3	ASCII octal (high byte first)
4	ASCII decimal (high byte first)
5	ASCII hex (high byte first)
6	ASCII floating point (high byte first)
7	IEEE float/double (low byte first)
8	<obsolete>
9	Allen-Bradley® PLC/3 floating point
10	VAX single precision floating point

For Options 7 and 9, the data is read as appropriate binary format.

Size

Required. Any numeric expression giving the number of digits in each datum. Size is measured in different ways for each format option (specified in previous parameter), as indicated in the following table:

Option	Size Meaning	Size Range
Binary types	Number of bits	1 – 32 bits
BCD	Number of 4-bit	1 – 8 digits

	digits	
ASCII types	Number of bytes	1 – 32 bytes
Float types	Precision	1 for single precision 2 for double precision

Skip

Required. Any numeric expression giving the number of buffer units to skip after each element is read. Units are bits for Options 0 and 1, BCD digits for type 2, and characters or bytes for all others.

Comments:

This function may only be used with buffers containing formatted numeric data. It reads the buffer, and places the data into module parameters. Normally, the module parameters will be variables. If a parameter is not a variable, then nothing will be assigned to that parameter, and this statement reads the next datum and continues on to the next parameter (if any). Illegal characters that are imbedded in the ASCII string stop further interpretation and are ignored. If an illegal character or the end of a buffer is encountered before a valid number, the remaining parameters are set to invalid.
The return value is the number of data read; this can be used as an error check (non-zero indicates an error).
This function is typically used in an I/O driver module to convert serial port or shared RAM data to VTScada variables.

Example:

In this example, suppose that a module is started with the statement (a.k.a. module call):

```
ReadIEEE(1, "PLC", 2, x, y, z);
```

and that the module ReadIEEE contains the following script:

```
If 1 Main;
[
  BuffToParm(Self() { Data goes in this module's parms },
```

```
           4 { Start at parm 4 (skip first 3 ) },
            response { Formatted buffer to be read },
            12 { Skip first 12 characters, 0 to 11 },
            NParm(Self()) - 3
            { Compute number of parms to read },
            7 { Format is IEEE floating point },
            2 { Double precision },
            1 { Skip 1 byte between each double });
]
```

This example reads IEEE double precision numbers from a buffer and places them in the module ReadIEE's parameters. The bytes 0 to 11 of response are skipped. Bytes 12 to 19 are converted from IEEE format and placed in x. Byte 20 is skipped (the skip parameter indicates 1 byte). Bytes 21 to 28 are converted and placed in y. Byte 29 is skipped. Bytes 30 to 37 are converted and placed in z. Reading ceases, (NParm(Self) - 3 equals 3, or three parameters have been read).

BuffToPointer

Description Converts a buffer of numeric data to array of pointers. This function reads from a formatted buffer containing numeric data, writes to locations specified by an array of pointers, and returns the number of elements read.

Returns Pointer array

Usage Script Only.

Function Groups Array, String and Buffer

Related to: BuffToArray | BuffToParm | New | PointerToBuff

Format BuffToPointer(ArrayElem, N, Buffer, Offset, Option, Size, Skip)

Parameters

ArrayElem

Required. Any array element giving the starting point in the pointer array. The subscript for the array may be any numeric expression.

N

Required. Any numeric expression giving the number of items to read. If N extends past the upper bound of the array dimension, this computation will "wrap-around" and resume at element 0, until N elements have been processed.

Buffer

Required. Any text or buffer expression to read.

Offset

Required. Any numeric expression giving the starting buffer position for the read in characters (bytes), starting at 0.

Option

Required. Any numeric expression that specifies the format of the buffer read. This can be one of:

Option	Buffer Format
0	Unsigned binary (low byte first)
1	Signed binary (low byte first)
2	BCD (binary coded decimal) (low byte first)
3	ASCII octal (high byte first)
4	ASCII decimal (high byte first)
5	ASCII hex (high byte first)
6	ASCII floating point (high byte first)
7	IEEE float/double (low byte first)
8	<obsolete>
9	Allen-Bradley® PLC/3 floating point
10	VAX single precision floating point

For Options 7 and 9, the data is read as appro-

priate binary format.

Size

Required. Any numeric expression giving the number of digits in each datum. Size is measured in different ways for each format option (specified in previous parameter), as indicated in the following table:

Option	Size Meaning	Size Range
Binary types	Number of bits	1 – 32 bits
BCD	Number of 4-bit digits	1 – 8 digits
ASCII types	Number of bytes	1 – 32 bytes
Float types	Precision	1 for single precision, 2 for double precision.

Skip

Required. Any numeric expression giving the number of buffer units to skip after each element is read. Units are bits for Options 0 and 1, BCD digits for type 2, and characters or bytes for all others.

Comments:

This function may only be used with buffers containing formatted numeric data. Illegal characters that are imbedded in the ASCII string stop further interpretation and are ignored. If an illegal character or the end of a buffer is encountered before a valid number, the remaining elements are set to invalid. As each item is read from the buffer, the result is stored to the location pointed to by the element of the array. If an ele-

ment is invalid, or doesn't contain a pointer, then nothing happens and processing continues with the next item/element.
The return value is the number of array elements read. This may be used as an error check – if the return value is not zero, then one or more array elements were not read.

Example:
Assume that the pointer array has been initialized as shown:

```
destination[0] = &stationFlow;
destination[1] = &stationPower;
destination[2] = &stationIntruder;
destination[3] = &stationOK;
destination[4] = &stationPressure;
```

Now execute the statement:

```
BuffToPointer(dDestination[0] { Starting element },
              5 { No. of elements to read },
              rResponse { Text expression to read },
              0 { Starting offset in buffer },
              2 { Read BCD format },
              4 { 4 BCD characters is 2 bytes },
              0 { Don't skip anything });
```

This reads 5 BCD format numbers into variables pointed to by elements 0 to 4. Each BCD number has 4 digits, which occupy 2 buffer bytes. This will set the variables stationFlow, stationPower, stationIntruder, stationOK, and stationPressure to the values read from the buffer.

BuffWrite

Description:	Writes formatted values to a buffer and returns the number of values not written.
Returns:	Numeric
Usage:	Script Only.
Function Groups:	String and Buffer
Related to:	BuffRead | FWrite | SetByte | SWrite | ArrayToBuff | MakeBuff
Format:	BuffWrite([N,] Buffer, Offset, Format, V1, V2, V3, ...)

Parameters:

Buffer

Required. Any text or buffer expression to be written. The buffer must already exist.

Buffers are created by certain functions, such as ArrayToBuff or MakeBuff, or by assignment of a text string such as "Hello." BuffWrite writes to an existing buffer, which is faster than creating a new buffer. Writing to an empty buffer has no effect.

Offset

Required. Any numeric expression giving the starting buffer position in characters or bytes for the write, starting at 0.

Format

Required. Any text expression giving the format of how the values (Vn parameters) are to be written. This format is similar, but not identical, to the C language format string for the printf function, whereby each of the Vn parameters in the statement is assigned to a % format specification in the order in which each appears in the list.

Note that like a standard text string, these format specifiers must also be enclosed by double quotes.

If a format specification appears for which there are no remaining V parameters, the format specification characters themselves are output to the stream exactly as they appear in the Format.

For the % format specifications, the following form applies (where the [ ] indicates optional elements):

%[-][+][SPACE][#][width][.precision]type
where
% (percent sign) is mandatory;
- (minus sign) causes the data to be left justified within the field (for binary types b and ASCII character types c, this option is ignored);
+ (plus sign) causes positive numbers to be prefaced with a + sign (negative numbers are unaffected). This allows easy alignment of positive and negative numbers in a printed column of numbers. For binary types b and non-numerical types, this option is ignored;
space represents the single space character, and is similar to the [+] option but places a single space rather than a plus sign in front of positive numbers (negative numbers are still unaffected). This allows alignment of a column of numbers without having to show all signs. For binary types b and non-numerical types, this option is ignored;
(hash mark) When used with the o , x , or X format, the # flag prefixes any nonzero output value with 0, 0x, or 0X, respectively.
width is a number that specifies the minimum number of characters to output. Numbers that require more characters than specified by the width value are truncated on output. If the number of characters in the number or string is less than width, blanks will be added to the left or right, depending upon whether the output is left or right justified (i.e. whether or not the [-] option has been specified) until the width is reached. For binary types b and ASCII character

types c, this option is ignored;
precision has a different meaning for each of the type options as follows:

- Integer types d, l, u, o, x, and X precision specifies the minimum number of digits to output. If the number contains fewer digits, leading zeroes will be added to the left of the number. If precision is 0, omitted, or if the decimal point appears without a number following it, the precision defaults to 1. The number is not truncated.
- Floating point types e and E precision specifies the number of digits after the decimal point. The last digit is rounded. The default precision in this case is 6. If the precision is 0 or if the decimal point appears without a number following it, no decimal point appears in the output.
- Floating point type f precision specifies the number of digits after the decimal point. The last digit is rounded. The default precision is 0. If the precision is explicitly 0, no decimal point is output. If a decimal point is output, at least one digit will be placed before the decimal point.
- Floating point types g and G precision specifies the maximum number of significant digits to be output. If no precision is specified, all significant digits are written.
- String type s precision specifies the maximum number of characters of the string to be output. If the string contains more characters than specified by the precision, the string is truncated and only the first characters are written. If the precision is not specified, all of the string characters are output.

- ASCII character type c The precision option is ignored.
- Binary type b The precision option is ignored.

x unsigned hex integer using "abcdef"

znnn Escape character where nnn is the 3-digit ASCII code

type is mandatory. The type specification must be one of those listed below.

Note: The case of the letter is important. Specifying a character for the type that is not in this list will result in all the characters following the % up to that point to be output exactly as they appear in the Format string.

Type	Meaning
nb	Binary format, where n is a number indicating the type of value (see below).
c	Single ASCII character (byte)
d	Signed decimal integer
e	Signed exponential; exponent key is "e".
E	Signed exponential; exponent key is "E".
f	Signed floating point.
g	e or f format, whichever is shorter.
G	E or f format, whichever is shorter.
h	Handle to a window.
i	Signed decimal integer.
o	Unsigned octal integer.
p	Pointer to a buffer.
s	Text string.

u	Unsigned decimal integer.
x	Unsigned hex integer using "abcdef".
X	Unsigned hex integer using "ABCDEF".

nb, Binary type For the format specification of %nb, where n specifies the type of number, n must be a single digit from one of the following choices. All are low-byte-first.

n	Value Type
0	Byte, unsigned
1	Signed short integer (2 bytes)
2	Signed long integer (4 bytes)
3	IEEE single precision float (4 bytes)
4	<obsolete>
5	IEEE double precision float (8 bytes)
6	<obsolete>
7	Unsigned short integer (2 bytes)
8	Unsigned long integer (4 bytes)

Note: Other options such as width and precision do not apply to the b type.

c, ASCII character type: This type is not representative of a single character in a string, but rather, represents single byte ASCII characters. Input values (the Vn parameter to which this format specification applies) must be integers in the range of 0 to 255 in order for the output to be a valid ASCII equivalent character. Strings are not acceptable input values. Note that the %c

format specifier behaves differently when used in an output statement such as BuffWrite than when used in an input statement, such as BuffRead.

d, Signed decimal integer:

e, Signed exponential: Exponent key is “e”

E, Signed exponential: Exponent key is “E”

f, Signed floating point

g, e or f formats: Whichever is shorter

G, E or F formats: Whichever is shorter

h, Window handle type: This type is used for building structures to be handed to DLLs and should be used by advanced users only.

p, Buffer pointer type: This type is also used for building structures to be handed to DLLs and should be used by advanced users only.

s, Text string type:

Plain text Text in the Format parameter is written exactly as it appears, with three exceptions:

- Percentage sign (%) Since format specifications for the Vn parameters are indicated by a percentage sign, to include an actual percentage sign as part of the Format parameter, precede it with a backslash character (i.e. \%).
- Backslash character (\) Since this is used to indicate special control characters such as line feed, carriage return, and form feed, to write a backslash as part of the Format parameter, use two backslash characters (i.e. \\).
- Quotation marks (") The entire test string is delimited by quotation marks, so to include a set of quotation marks as part of the Format parameter, use a set of quotations marks (i.e. "").

Control characters In order to encode certain control characters as part of the Format parameter, one of two methods may be used. The first is to use a backslash character followed by one of the single character codes listed below to produce the desired result (notice that the letters must be lower case):

Code	Meaning
\b	Backspace
\f	Form feed
\n	Line feed
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab

\nnn In addition to the above predefined codes, \nnn may be used, where nnn is a three digit integer in the range of 0 to 255 specifying a certain ASCII character. If the number contains less than three digits, the leading spaces must be padded with zeroes; this is not the case with the previously listed single character control characters. For example, to include the one byte ASCII character G in the output, you could place its decimal equivalent of 71 in the Format string as \071.

u, Unsigned decimal integer,

x, Unsigned hex integer using “abcdef”

X, Unsigned hex integer using “ABCDEF”

Offset is any numeric expression giving the starting buffer position in characters or bytes for the

write, starting at 0.

Comments:

In early versions of VTS (WEB), there was a numeric leading parameter, N. This should not be included in any new code.

If one of the values to be written is outside of the range of the type indicated by the format specifier, a 0 is written. If the value to be written is invalid, nothing is written for most format specifiers, with the exception of %nb, which will write a 0 in the place of the invalid. Invalidity of the output values does not preclude execution of this function.

This function returns the number of Vn parameters not written to the buffer; a 0 return value indicates success. Variables that contain invalid values that were not written due to their invalidity do not increment this count. An invalid return value indicates an error with one of the parameters.

This function can be used to format strings for display, or for message packets as part of the driver toolkit.

Example:

```
If ! Valid(buff);
[
  buff = MakeBuff(100, 0) { Create the buffer };
  BuffWrite(buff { Buffer },
            0 { Starting offset },
            "A=%3.2d\r\nB=%6.2f\r\n%8.3s\r\n%c\r\n\033"  { Format
string },
            2, 2/3, "finished", 33 { Values to be written });
]
```

This would set buff as follows :

```
A= 02
B= 0.67
fin
!
!
```

BuildDelete

(ODBC Manager Library)

Description: Builds SQL Delete statements based on arrays of field names and values. Made to be called as a subroutine only.

Returns: Text (the SQL statement)

Usage: Script Only.

Related to: BuildInsert | BuildSelect | BuildUpdate

Format: \ODBCManager\BuildDelete(TableName, WhereFields, WhereOperators, WhereValues, WhereSQLDataTypes, WhereAND, dbType)

Parameters:

TableName

Required. Any text expression for the table name to delete records from.

WhereFields

Required. May be a simple value or a one-dimensional array. Provides the field names for the WHERE selection clause

WhereOperators

Required. May be a simple value or a one-dimensional array. Provides the operators for the WHERE selection clause

WhereValues

Required. Any SQL data type. May be a simple value or a one-dimensional array. Values for the WHERE selection clause

WhereSQLDataTypes

Required. Values indicating the data type of the insert values. Should be a simple value or an array matching the WhereFields parameter. Refer to Data Type Codes used in the ODBC Manager for a list of the value codes.

WhereAND

Required. Any expression that evaluates to a Boolean true or false.
If set to true (non-zero) then the components of the

WHERE clauses are to be ANDed together.
If false (0) an OR is used between the sub clauses.

dbType

Required numeric value, indicating the type of this DB connection.

DBType	Meaning
0	MS SQL
1	MS Access
2	Oracle
3	MySQL
4	SyBase

Comments:

This module is a member of the ODBCManager Library, and must therefore be prefaced by \ODBCManager\, as shown in "Format" above.
If the WhereFields parameter is invalid, the SQL statement returned will be "DELETE FROM [Tablename]". The ExecuteQuery function includes a check that will prevent a delete statement without a where clause from running.

BuildFullName

Description If a namespace and namespace delimiter are being used, returns the full, namespace-qualified name of the specified account.

Returns String

Usage Script Only.

Related to: GetAccountID | GetAccountInfo | GetGroupName | GetFullName | GetUserName | IsLoggedOn | IsSecured | IsSuspended | SecurityCheck | UIErrorToText | See also, "Security NameSpaces" in the VTScada Programmer's Guide.

Format BuildFullName(AccountName [, GroupName]);

Parameters

AccountName

The name of an account to generate the full name for.

GroupName

The name of a namespace (group) to use when building the full name.

Comments If the configuration setting NameSpaceDelimiter is valid, returns the full, namespace-qualified name of the specified account.
If the GroupName parameter is Invalid or not specified, or if the account is a member of the root namespace, or the configuration setting NameSpaceDelimiter is Invalid, the return value is the account name, as passed in.

BuildInsert

(ODBC Manager Library)

Description: Builds SQL Insert statements based on arrays of field names and values. Made to be called as a subroutine only.

Returns: Text or ODBCQuery structure.

Usage: Script Only.

Related to: BuildDelete | BuildSelect | BuildUpdate

Format: \ODBCManager\BuildInsert(TableName, InsertFields, InsertValues, SQLDataTypes, dbType)

Parameters:

TableName

Required. The name of the table into which data will be inserted.

InsertFields

Required. May be a simple value or a one-dimensional array. Field names matching the InsertValues array of data to be inserted

InsertValues

Required. Any SQL data type. May be a simple value or a one-dimensional array. Provides the new value(s) for the fields in the matching InsertFields parameter.

SQLDataTypes

Required. Values indicating the data type of the insert values. Should be a simple value or an array matching the InsertFields parameter. Refer to Data Type Codes used in the ODBC Manager for a list of the codes.

dbType

Required numeric value, indicating the type of this DB connection.

DBType	Meaning
0	MS SQL
1	MS Access
2	Oracle
3	MySQL
4	SyBase

Comments:

This module is a member of the ODBCManager Library, and must therefore be prefaced by \ODBCManager\, as shown in "Format" above.
Returns the SQL INSERT statement as a text string, unless long binary data is involved, in which case an ODBCQuery structure is returned. The format of the structure is as follows:

```
ODBCQuery STRUCT [
  QueryString;
  Parameters;
];
```

BuildSelect

(ODBC Manager Library)

Description: Builds SQL selection queries using supplied field names and values. Made to be called as a subroutine only.

Returns: Text (the SQL query)

Usage: Script Only.

Related to: BuildInsert | BuildDelete | BuildUpdate

Format: \ODBCManager\BuildSelect(SelectFields, TableName, WhereFields, WhereOperators, WhereValues, WhereSQLDataTypes, WhereAND, OrderFields, Qualifier)

Parameters:

SelectFields

Required. Text array of field Names to read

TableName

Required. May be a simple value or a one-dimensional array. The name(s) of the table(s) that will be queried.

WhereFields

Required. May be a simple value or a one-dimensional array. Field names for WHERE clause

WhereOperators

Required. May be a simple value or a one-dimensional array. Operators for WHERE clause

WhereValues

Required. Any SQL data type. May be a simple value or a one-dimensional array. Values for WHERE clause

WhereSQLDataTypes

Required. Values indicating the data type of the insert values. Should be a simple value or an array matching the WhereFields parameter.
Refer to Data Type Codes used in the ODBC Manager for a list of the codes.

WhereAND

Required. Can be any expression that evaluates to a Boolean true or false.
If set to true (non-zero) then the components of the WHERE clauses are to be ANDed together.

If false (0) an OR is used between the sub clauses.

OrderFields

Required. May be a simple value or a one-dimensional array. Provides the field names for ORDER BY clause

Qualifier

Required. SQL Qualifier such as "top 100", "unique", etc

Comments:

This module is a member of the ODBCManager Library, and must therefore be prefaced by \ODBCManager\, as shown in "Format" above.

BuildUpdate

(ODBC Manager Library)

Description: Builds SQL UPDATE statements using supplied field names and values. Made to be called as a subroutine, but will function as a called module.

Returns: Text

Usage: Script Only.

Related to: BuildInsert | BuildSelect | BuildDelete

Format: \ODBCManager\BuildUpdate(TableName, UpdateFields, UpdateValues, SQLDataTypes, WhereFields, WhereOperators, WhereValues, WhereSQLDataTypes, WhereAND, dbType)

Parameters:

TableName

Required. Any expression for the name of the table to be updated.

UpdateFields

Required. May be a simple value or a one-dimensional array. Provides the field names to be updated

UpdateValues

Required. Any SQL data type. May be a simple value or

a one-dimensional array. Provides the new values for the fields

SQLDataTypes

Required. Values indicating the data type of the update values. Should be a simple value or an array matching the UpdateFields parameter. Refer to Data Type Codes used in the ODBC Manager for a list of the numeric codes.

WhereFields

Required. Any expression or array of the field names for WHERE clause

WhereOperators

Required. May be a simple value or a one-dimensional array. Operators for WHERE clause

WhereValues

Required. May be a simple text value or a one-dimensional array of text. Values for WHERE clause

WhereSQLDataTypes

Required. Values indicating the data type of the insert values. Should be a simple value or an array matching the WhereFields parameter. Refer to Data Type Codes used in the ODBC Manager for a list of the numeric codes.

WhereAND

Required. Can be any expression that evaluates to a Boolean true or false. If set to true (non-zero) then the components of the WHERE clause are to be ANDed together. If false (0) an OR is used between the sub clauses.

dbType

Required numeric value, indicating the type of this DB connection.

DBType	Meaning
0	MS SQL
1	MS Access
2	Oracle
3	MySQL
4	SyBase

Comments:

This module is a member of the ODBCManager Library, and must therefore be prefaced by \ODBCManager\, as shown in "Format" above. Returns the SQL UPDATE statement as a text string unless binary parameters are involved, in which case an ODBCQuery structure is returned. The format of the structure is as follows:

```
ODBCQuery STRUCT [
  QueryString;
  Parameters;
];
```

C Functions

The sections that follow identify all VTScada functions beginning with "C".

Call

Description: Starts an instance of the module specified by its first parameter.

Returns: Nothing

Usage: Script or steady state.

Function Groups: Compilation and On-Line Modifications, Basic Module

Related to: FindVariable | Self

Format: Call(Module [,Parm1, Parm2, Parm3, ...])

Parameters:

Module

Required. Any expression for the module that is to be started.

Parm1, Parm2, Parm3, ...

Optional. Expressions to be passed to the module that is being started.

Comments: The additional parameters that are specified are passed to the started module as parameters to the call.

Example:

```
ZEditField(10, 40, 110, 10, mod, 32, 0, 1);
If Valid(mod) && ! Valid(x);
[
  x = FindVariable(mod, Self(), 0, 1);
]
...
Call(x);
```

This section of code enables the user to enter a module name into the edit field, which will then be located by the script and started by the Call statement. Notice that in this particular example, no parameters are passed to the module, however, more edit fields could be created to accept variables or values to use as parameters to the module by entering them in the Call statement.

CalledInstances

Description: Returns the object values of module instances that are called by a particular module.

Returns: Object(s)

Usage: Script Only.

Function Groups: Advanced Module, Compilation and On-Line Modifications

Related to: ChildInstances | Self | GetInstance | Instance | NumInstances | Valid

Format: CalledInstances(Object [Options])

Parameters:

Object

Required. An object value of the module instance for which to get the number of called instances.

Options

Optional. Any numeric expression that defines which modules are to be included in the returned set.

The value for this parameter is formed by adding together the values from the following table:

Options	Bit Number	Description
1	0	Include all modules, even if they are in the same window as their caller; false to only include (root) modules for modules in separate windows.
2	1	Recurse into called modules of called modules; false to only include modules directly called by the Object parameter.
4	2	Group all instances of the same module into an array and store that array of

object values in the element of the returned array, instead of the object value.

Comments: The return value is an array of objects which are called from Object. If no instances are called from Object, Invalid is returned.

Example:

```
i = -1;
...
If i == -1;
[
  numCalled = CalledInstances(Self());
  i++;
]
If Valid(numCalled[i]);
[
  Slay(numCalled[i], 0);
  i++;
]
```

The first If statement and script creates an array of object values to all instances called by the current module; the second If and script slays all of these instances. Notice that in the case of the first script, the action trigger must be based on i rather than on a Valid test of numCalled; if this were not the case and ! Valid(numCalled) was used instead, then, if there were in fact no instances called from this module, CalledInstances would return Invalid, and an "if one" (infinite loop) condition would occur.

Caller

Description: Takes a given object value for a module and returns the object value of the module by which it was called.

Returns: Object

Usage: Script or steady state.

Function Groups: Compilation and On-Line Modifications, Basic Module

Related to: FindVariable | Self

Format: Caller(Object)

Parameters:

Object

Required. An object value that specifies the instance whose caller is desired.

Example:

```
obj = Caller(Self());
```

This assigns the pointer to the caller of the current module to variable Obj.

CallerID

Description: Takes a specified modem stream and returns the caller ID from the telephone system.

Returns: Text

Usage: Script Only.

Function Groups: Modem

Related to:

Format: CallerID(ModemStream)

Parameters:

ModemStream

Required. The modem stream from which you wish the caller ID returned.

Comments: A modem that supports caller ID as provided by the local phone service is required.

CancelCall

Modem Manager

Description: This subroutine removes a queued call or abandons a call that is in-progress.

Usage: Script Only.

Related to: MakeCall

Format: \ModemManager\CancelCall(Tag [, HangUp, NoCancel, Silent]);

Parameters:

Tag

Any text expression that identifies the tag that originally requested the call.

HangUp

An optional Boolean parameter that specifies whether or not to hang-up on an active call. The default is to allow an active call to proceed.

NoCancel

An optional Boolean parameter that, if set, will remove the call from the internal queue, but not terminate the call.

Silent

An optional Boolean parameter indicating the logging is not required if set TRUE.

Comments: As described under MakeCall, once a tag has requested that a call be made, the Modem Manager takes responsibility for the control of the call, and indicates progress in the tag's DataPort variable. If the tag decides at any point before the Modem Manager has completed (or abandoned) the call that it no longer requires the call, then the tag must call CancelCall.

CanEditDoc

Description: Returns an indication as to whether or not the document

for the given module can be edited.

Warning: This function should be used by advanced programmers only.

Returns: Boolean

Usage: Script Only.

Function Groups: Compilation and On-Line Modifications, Advanced Module

Related to: ResyncDoc

Format: CanEditDoc(Module [, ErrCode])

Parameters:

Module

Required. Any module value that specifies the document that you wish to modify.

ErrCode

Optional. Is set to a non-zero value when CanEditDoc returns a 1. The return value is a collection of 3 bits:

ErrCode	Bit No.	Description
1	0	TRUE if not available (001)
2	1	TRUE if read only (010)
3	2	TRUE if file is out of sync (100)

Comments: This function returns true if the document for the given module can be modified. The function will check to see if the date and time for the document file match that of the run file. It will return false if the document does not exist, if the dates and times are out of sync, or if the document is "Read Only".

Example:

```
If Watch(1, CurrentWindow());
[
  IfElse(CanEditDoc(CurrentWindow()),
        editFlag = 1,
        editFlag = 0);
]
```

This script will be executed as soon as its state becomes active, and then every time the mouse moves over a new window; the flag called editFlag will indicate whether or not editing can be done in the window that the mouse is over.

CaptureImage

Description: Creates an image handle from a GUIStretch operation

Returns: Bitmap handle

Usage: Script Only.

Function Groups: Graphics

Related to: SaveImage |

Format: CaptureImage(Object, Left, Bottom, Right, Top);

Parameters:

Object

The object whose image is being captured

Left

The left coordinate of the capture.

Bottom

The bottom coordinate of the capture.

Right

The right coordinate of the capture.

Top

The top coordinate of the capture.

Comments: Creates an image capture of anything drawn by the object

or its children within the provided (optional) coordinates. This image is stored in a bitmap handle identical to the output of the MakeBitmap function, meaning that it can be modified, displayed or saved.

Examples:

```
{ GUIStretch calling a module }
GUITransform(296, 736, 469, 563,
       1, 1, 1, 1, 1 { Scaling },
       0, 0 { Movement },
       1, 2 { Visibility, Reserved },
       0, 0, 0 { Selectability },
       Obj = MyModule("GUIStretch of MyModule"));

{ Capture an image of the module every second }
If TimeOut(1, 1);
[
   CapturedImage = CaptureImage(Obj);
]
```

CaptureSettings

Description: Gathers a single property value or an accumulated section and returns the result in a tabular format.

Returns: Object

Usage: Script Only.

Function Groups: Configuration Management

Related to: GetINIProperty |

Format: LayerRoot\CaptureSettings(Section, ValueName, pResult, CallerHasLock)

Parameters:

Section

Required. The name of the section where the property will be found.

ValueName

Required. The name of the property to return. Set to Invalid to retrieve the entire section.

pResult

Required. A pointer to a variable, in which the retrieved property or properties will be returned.

CallerHasLock

Boolean. Set to TRUE if the caller holds the semaphore.

Comments: This function gathers settings data from an active cache that keeps up-to-date inheritance information across all ancestor layers back to the setup.INI file for the VTScada System layer. This is the preferred method for inquiring about the current value of a setting that is not stored in a "Code" object variable ("Code" variables are created for settings in the [SYSTEM], [LABELS], and [AREAS] sections). This is an asynchronous operation which in most cases executes very quickly, its return value is an object which becomes invalid when the op completes. The caller must not slay itself while this object is valid.

Comments are not included in the output, but hidden values are. The settings accumulation is stored in a Layer-level variable named CaptureCache. This variable is refreshed on the next call after a settings file changed as reported by LayerSettingsMod\LayerNotify via the RecaptureSettings flag. If this flag isn't set then CaptureSettings simply reads the requested value from the cache without acquiring the lock. The check is performed in a critical section to prevent result corruption due to settings changes on other threads.

Examples:

```
EmailProperties = \LayerRoot\CaptureSettings("System", "OutBoundE-
mailSettings", &EmailSettings, 0);
```

Case

Description: Selects one of a set of parameters for execution and

returns its return value.

Returns: Numeric

Usage: Script only.
May be used in optimized Tag Parameter Expressions.

Function Groups: Logic Control

Related to: ?: (If Else) | Cond | Execute | IfElse | IfThen

Format: Case(Index, P0, P1, P2)

Parameters:

Index

Required. Any numerical expression giving the parameter number to execute. If this value is 0, the parameter executed is P0.

P0, P1, P2, ...

Required. The statements, one of which will be executed as specified by the value of Index.
At most one statement is executed. These parameters are typically Execute statements.

Comments: The return value of this function is the return value of the parameter executed, or invalid if no parameter is executed.

Example:

```
If 1 Main;
[
  Case(pumpNum { Number varies from 0 to 3 },
              { 0 }pumpDesc = "Pump 0",
              { 1 }Execute(pumpDesc = "Pump 1", Diesel = TRUE),
              { 2 }pumpDesc = "Pump 2",
              { 3 }pumpDesc = "Pump 3");
]
```

This sets the pumpDesc variable according to pumpNum and in the second case, uses the Execute function to accomplish more than one task based on the value of pumpNum. If setting pumpDesc was all that needed to be done, the statement could have been shortened to:

```
If 1 Main;
[
  pumpDesc = Case(pumpNum,
                { 0 } "Pump 0",
                { 1 } "Pump 1",
                { 2 } "Pump 2",
                { 3 } "Pump 3");
]
```

Cast

Description Takes a value and returns a different type of value, if possible.

Returns Numeric

Usage Script or steady state.

Function Groups Compilation and On-Line Modifications, Variable

Related to: ValueType

Format Cast(Val, Type)

Parameters

Val

Required. Any variable name.

Type

Required. A VTScada Value Types - Numeric Reference indicating what type of value should be returned.

Comments This function performs a type-cast, changing one type of value to another. Note that values that are converted to integers are truncated, rather than rounded.
If a stream longer than 65,523 characters is cast to a text string, it will be truncated at 65,523 characters.

Example:

```
xFloat = 2.61;
xInt = Cast(xFloat, 1);
```

The value of xInt will be set to 2.

Ceil

Description: Returns the smallest integer greater than or equal to a number (the ceiling).

Returns: Numeric

Usage: Script or steady state.

Function Groups: Rounding Math

Related to: Int | Step

Format: Ceil(X)

Parameters:

X

Required. Any numeric expression for which the ceiling should be determined.

Comments: This function performs function similar to that of the Int function, except that it goes to the next highest number.

Example:

```
a = Ceil(1.00);
b = Ceil(1.01);
c = Ceil(1.99);
```

The values of a, b and c will be 1, 2 and 2 respectively.

Change

Description: Returns a true when the value of the first parameter changes by at least the value of the second parameter.

Returns: Boolean

Usage: Steady State only.

Function Groups: Variable

Related to: DeadBand | Edge | Latch | Toggle | Save

Format: Change(Value, MaxLimit)

Parameters:

Value

Required. Any numeric expression giving the value to check for the change.

MaxLimit

Required. Any numeric expression giving the amount by which Value must change for the function to be true. The change in Value must be strictly greater than MaxLimit for the Change function to be true.
If MaxLimit is less than zero, the function will always be true.

Comments:

The change specified by MaxLimit is the absolute value of the change so the Value is checked for an increase or decrease by this amount.
The initial value used in the comparison for the change is the value when the function is first executed upon entering a state or when the parameters become valid. This initial value is reset by functions that reset their parameters (i.e. Latch, Toggle, and Save) and by action triggers.
Note that Value must change from a valid value to another valid value; changing to or from an invalid value does not trigger a Change.

Example:

```
Save(0, 0, 0, 0, 1, 0, 0 { Save 1 float value to disk },
     1000 { Number of records },
     50 { Buffer 50 records },
     "G:\DATA\SETPT.DAT" { file name },
     Change(sp, 0) { Trigger - any changes in sp },
     sp { Setpoint value to log });
```

This shows how to use a Change function as a trigger for a Save statement to log data whenever the datum changes; notice that by using a 0 as the MaxLimit value, all changes no matter how small are registered. When the Save statement is triggered (when data are logged), the Change statement is reset to wait for another change.
An example of how Change is used is as an action trigger follows:

```
If Change(x, 0.5);
[
  ...
]
```

The script will execute once every time x changes by more than 0.5. The Change function is reset when it is used in a action trigger and it becomes true. Suppose the following happens to x:

```
x (initial value) = 3.4
x (changes to) = 3.5 -> Nothing happens yet
x (changes to) = 3.9 -> Action triggers, script executes,
Change is reset
```

Change now waits for x < 3.4 or x > 4.4

```
x (initial value) = 4.1
x (changes to) = 4.2 -> Nothing happens yet
x (changes to) = 5.1 -> Action triggers, script executes,
Change is reset
```

Change now waits for x < 4.6 or x > 5.6

Related Information:

See: "Latching and Resetting Functions" in the VTScada Programmer's Guide

ChangePersistentSize

Description:	Changes the space allocated in the persistent value (.VAL) file for a variable.
Warning:	This function should be used by advanced programmers only since irrevocable alteration of your application may occur.
Returns:	Nothing
Usage:	Script Only.
Function Groups:	Compilation and On-Line Modifications, Variable
Related to:	AddVariable | FindVariable | MakeNonPersistent | MakePersistent | PersistentSize
Format:	ChangePersistentSize(Variable, Size)
Parameters:	

Variable

Required. Any expression for the variable value. This value is usually returned from a call to AddVariable or FindVariable.

Size

Required. The new size for the persistent variable.

Comments: This statement will only work on variables that are already persistent. It will adjust the size of allocated space in the persistent value (.VAL) file for the owning module.

Example:

```
If 1 Main;
[
  ChangePersistentSize(FindVariable("runningHrs",
                       Self(), 0, 1), 40);
]
```

This changes the persistent size for the variable runningHrs (if it was already defined as persistent).

CharCount

Description: Returns the number of bytes in a section of a buffer that matches a search character.

Returns: Numeric

Usage: Script or steady state.

Function Groups: String and Buffer

Related to: Locate | Replace

Format: CharCount(Buffer, Offset, N, Match)

Parameters:

Buffer

Required. Any text expression giving the buffer to search.

Offset

Required. Any numeric expression giving the starting buffer position for the CharCount, starting at 0.

N

Required. Any numeric expression giving the number of bytes to include in the search.

Match

Required. Any numeric expression giving the byte (usually an ASCII code) for which to search. Match must be in the range of 0 to 255.

Comments: Offset + N must be less than or equal to the buffer length, or the return value will be invalid.

Example:

```
num = CharCount("abcdefABCDEFc" { Search buffer },
                0 { Start of the buffer },
                13 { Search the entire buffer },
                0x63 { Find the character "c" });
```

Num is set to 2 (the two lower case "c"s match but the upper case "C" doesn't).

CheckBox

(System Library)

Description: Draws a check box with (optional) label.

Returns: Nothing

Usage: Steady State only.

Function Groups: Graphics

Related to: Droplist | GridList | Listbox | Spinbox

Format: \System\CheckBox(X1, Y1, X2, Y2, Variable [, Label, BoxOnLeft, Alignment, FocusID, BGColor, FGColor])

Parameters:

X1

Required. Any numeric expression giving the X coordinate on the screen of one side of the check box and its label. The smaller of X1 and X2 will always be to the left

Y1

Required. Any numeric expression giving the Y coordinate on the screen of either the top or bottom of the check box. The smaller of Y1 and Y2 will always be the top.

X2

Required. Any numeric expression giving the X coordinate on the screen of the side of the check box and its label opposite to X1.

Y2

Required. Any numeric expression giving the Y coordinate on the screen of the top or bottom of the check box, whichever is the opposite to Y1.

Variable

Required. The variable whose value is toggled by the check box.

Label

Optional. Any text expression to be used as a label with the check box. The default value is a blank label.

BoxOnLeft

Optional. Any logical expression. If true (non-0) the check box will appear to the left of the label.
If false (0) it will be to the right.
The default value is true.

Alignment

Optional. Any numeric expression that sets the alignment of the check box and its label accord-

ing to one of the following options:
The default value is 0.

Value	Horizontal Alignment	Vertical Alignment
0	Left	Top
1	Right	Top
2	Full	Top
3	Left	Centered
4	Right	Centered
5	Full	Centered
6	Left	Bottom
7	Right	Bottom
8	Full	Bottom

FocusID

Optional. Any numeric expression for the focus number of this graphic.
If this value is 0, the check box will display its current setting, but will not be able to be set and will appear grayed out. The default value is 1.

BGColor

Optional. Any numeric expression for the background color of the control. No default value.

FGColor

Placeholder for the foreground color of the control. Not currently implemented.

Comments:

This module is a member of the System Library, and must therefore be prefaced by \System\, as shown in the "Format" section. If you are developing a script application, use "System\..." rather than "\System\..." in the function

call.
The variable must be initialized to 0 or 1 or the check box will not be able to be toggled.
The size of the check box is constant, with X1, Y1and X2, Y2 defining the position of the check box and its label.
For any optional parameter that is to be set, all optional parameters preceding the desired one must be present, although they may be invalid.

Examples:

```
System\CheckBox(90, 220, 160, 205 { Location of check box },
                visibility { Variable to change },
                "Visible Switch" { Label },
                0 { Box not on left },
                5 { Full, center align },
                4 { Focus ID });
                System\CheckBox(90, 220, 160, 205 {
                     Location of check box },
                transparency { Variable to change },
                "Transparent" { Label },
                Invalid { Use default },
                0 { Left, top align });
{ Use default focus ID }
```

CheckFileExist

(System Library)

Description: This subroutine checks for the existence of the specified file.

Returns: Boolean

Usage: Script only.
May be used in optimized Tag Parameter Expressions.

Function Groups: File I/O

Related to: CheckPathExist | CopyDir

Format: System\CheckFileExist(FileName)

Parameters:

FileName

Required. Any expression for the name of the file

whose existence you wish to verify.

Comments: This subroutine checks for the existence of the file specified by FileName. It returns 1 if the specified file exists, or 0 if the specified file does not exist.

CheckPathExist

(System Library)

Description: This subroutine checks for the existence of the specified path.

Returns: Boolean

Usage: Script only.
May be used in optimized Tag Parameter Expressions.

Function Groups: File I/O

Related to: CheckFileExist | CopyDir

Format: System\CheckPathExist(Path)

Parameters:

Path

Required. Any expression for the path whose existence you wish to verify.

Comments: This subroutine checks for the existence of the path specified in Path. It returns 1 if the specified path exists, or 0 if the specified path does not exist.

CheckTagGroup

Description: Returns TRUE or FALSE according to whether a tag is in the specified group.

Returns: Boolean

Usage: Script Only.

Function Groups: Variable

Related to:

Format: \CheckTagGroup(TagModule, TagGroup)

Parameters:

TagModule

Required. A tag type module or instance.

TagGroup

Required. The name of the tag group to check.

Comments: If the tag is a member of the group, the function will return, TRUE. A list of groups can be found in the topic, Tag Groups.

Examples:

```
IfThen(\CheckTagGroup(TagModule, "Analogs"),
   ...
);
```

ChildDocs

Description: Gets the module values for the root and all descendent modules that match the conditions defined by the second parameter. May also be called as "Child_Docs".

Returns: Pointer to a one dimensional array

Usage: Script Only.

Function Groups: Compilation and On-Line Modifications, Advanced Module

Related to: ModuleFileName | RUNFileName

Format: ChildDocs(Module [, Filter])

Parameters:

Module

Required. The module to search.

Filter

Optional. Any numeric expression that defines which modules are to be included in the returned set. Filter is formed by adding together the values from the following table. Defaults to 2 if missing or invalid.

Filter	Bit No.	Description
1	0	Include all modules, even if they are in the same file as their parent; false to only include modules with external files (subject to bits 2 and 5).
2	1	Recurse into submodules; false to include only immediate children.
4	2	Include only modules whose .RUN files are out of sync with their .SRC files.
8	3	Include the root module in the list, subject to bits 2 and 5.
16	4	Don't recurse if the module is added to the list.
32	5	Include only modules whose .SRC file exists (i.e. not just a .RUN file).

Comments: This function is used by the compiler subsystem. It returns a pointer to a single dimensioned array.

Example:

```
If 1 Main;
[
   AllList = ChildDocs(Scope(Self(), "Graphics"), 1 + 2 + 8);
]
```

The variable allList will be set to an array containing all module values that are ancestors of module Graphics, including Graphics itself.

ChildInstances

Description Returns the object values of module instances that are children of a particular module instance (i.e. all objects whose parent is a specified object).

Returns Object array

Usage Script Only.

Function Groups Basic Module

Related to: CalledInstances | GetInstance | Instance | NumInstances | Self | Valid

Format ChildInstances(Object [, Options])

Parameters

Object

Required. An object value of the module instance for which to get the number of child instances.

Options

Required. Any numeric expression that defines which modules are to be included in the returned set. Options is formed by adding together the values from the following table:

Options	Bit Number	Description
1	0	Include all modules, even if they are in the same window as their caller; false to only include (root) modules for modules in separate windows.

2	1	Recurse into child module instances of children; false to only include modules whose immediate parent is the module instance referenced by the Object parameter.
4	2	Group all instances of the same module into an array and store that array of object values in the element of the returned array, instead of the object value.

Comments The return value is an array of objects that are called from Object. If no instances are called from Object, Invalid is returned.

Example:

```
i = -1;
...
If i == -1;
[
  numKids = ChildInstances(Self());
  i++;
]
If Valid(numKids [i]);
[
  Slay(numKids[i], 0);
  i++;
]
```

The first If statement and script creates an array of object values to all instances that are children of the current module; the second If and script slays all of these instances. Notice that in the case of the first script, the action trigger must be based on i rather than on a Valid test of numKids; if this were not the case and !Valid(numKids) was used instead, then, if there were in fact no child instances of this module,

ChildInstances would return invalid, and an "if one" (infinite loop) condition would occur.

Circle

Note: Deprecated. Do not use in new code.

Description: Draws a circle on the screen.

Returns: Nothing

Usage: Steady State only.

Function Groups: Graphics

Related to: Arc | Ball | Box | Ellipse | GUIEllipse

Format: Circle(X, Y, Radius, Color, Width)

Parameters:

X

Required. Any numeric expression giving the X coordinate for the center of the circle on the screen.

Y

Required. Any numeric expression giving the Y coordinate for the centre of the circle on the screen.

Radius

Required. Any numeric expression giving the radius of the circle specified in units of X screen coordinates.

Color

Required. Any numeric expression giving the color of the circle.

Width

Required. Any numeric expression giving the width of the circle wall in units of X screen coordinates. The width is always rounded to result in an odd number of pixels on the screen. The minimum width displayed will be 1 pixel.

Comments: This statement has been superseded by the GUIEllipse function and is maintained for backwards compatibility only.
As of version 11, this is now drawn in the same z-order as other graphics, making it similar to the z-graphics functions.

Example:

```
Circle(XLoc(), YLoc() { Center coordinates follow mouse },
       100 { Radius in screen coordinates },
       11 { Light cyan color },
       1 { Line width in screen coordinates });
```

This draws a circle around the mouse cross-hairs that moves wherever the mouse is moved.

CleanModule

Description: Removes the flag that marks when a module that has been changed programmatically and would therefore have its changes saved to disk were this flag not cleared.

Returns: Nothing

Usage: Script Only.

Function Groups: Advanced Module

Related to:

Format: CleanModule(Module)

Parameters:

Module

Required. Any expression giving a module to clean.

Comments: When a module's code is changed through VTScada script (for example, by using a function such as AddVariable) a flag is set in the engine to indicate that the associated script file must be updated before shutdown. This function clears that flag, preventing the update.
This function can be used to make transitory code changes

(such as, by the Expression Manager) or to reverse code changes that have been undone (page editing).

Note: This function requires a module handle as its sole parameter such as the return value of a LoadModule function.

Example:

```
X = LoadModule(…);
{ … module is modified by code … }
CleanModule(X);
```

ClearModule

Description Deletes the contents (all variables and states) of a module without removing the module itself.

Warning This function may cause irrecoverable alteration of your application. It should be used only by advanced programmers

Returns Nothing

Usage Script Only.

Function Groups Compilation and On-Line Modifications, Advanced Module

Related to: ClearState | DeleteModule

Format ClearModule(Module)

Parameters

Module

Required. Any expression giving a module to clear.

Comments If there are any instances of the module running, ClearModule does nothing and returns.

ClearState

Description: Deletes all of the statements in a state.

Warning: This function may cause irrecoverable alteration of your

application. It should be used only by advanced programmers.

Returns: Nothing

Usage: Script Only.

Function Groups: Compilation and On-Line Modifications, States

Related to: ClearModule | DeleteState

Format: ClearState(State)

Parameters:

State

Required. Any expression giving the code value of the state to delete.

Comments: This statement is used to delete the contents of a state without removing the state itself or altering the document file.

ClearVarMetaData

Description: The opposite of SetVarMetaData, this statement removes all metadata associated with a variable.

Returns: Nothing

Usage: Script Only.

Function Groups: Dictionary

Related to: GetVarMetadata | SetVarMetadata

Format: ClearVarMetadata(Var)

Parameters:

Var

Required. Any expression giving the variable whose metadata base value is to be removed.

Comments: none

Click

Description: Returns an indication of whether or not the mouse pointer is within a specified screen area and a particular button combination is being pressed.

Returns: Boolean

Usage: Steady State only.

Function Groups: Graphics, Locator, Window

Related to: LocSwitch | Pick | SetXLoc | SetYLoc | Target | WinLocSwitch | WinXLoc | WinYLoc | XLoc | YLoc

Format: Click(X1, Y1, X2, Y2, Button)

Parameters:

X1

Required. Any numeric expression giving the X coordinate on the screen of one side of the screen area ("target").

Y1

Required. Any numeric expression giving the Y coordinate on the screen of either the top or bottom of the screen area ("target").

X2

Required. Any numeric expression giving the X coordinate on the screen of the side of the "target" opposite to X1.

Y2

Required. Any numeric expression giving the Y coordinate on the screen of either the top or bottom of the "target", whichever is the opposite to Y1.

Button

Required. Any numeric expression giving the locator button combination that will cause the Click

function to return a true value when the locator cursor is within the "target" screen area. The codes for this parameter are as follows:

Button	Button Locator
0	No buttons
1	Right button
2	Middle button
3	Right and middle buttons
4	Left button
5	Left and right buttons
6	Left and middle buttons
7	All three buttons

If the value is:

- multiplied by 8:
  the meaning for multiple buttons pressed becomes "OR", rather than "AND". For example, to accept any button on a 2 or 3 button mouse, use 56 (8 * 7); to accept the left mouse button, regardless of whether or not the right button is pressed, use 32 (8 * 4).
- increased by 64:
  the function will become true when the mouse buttons are released, rather than when they are pressed.
- increased by 128:
  the buttons must be double-clicked

Comments: This function returns true if the locator position is within the boundaries of the "target" as defined by (X1,Y1) - (X2,Y2), and the locator button value matches the Button parameter. If the locator is not

installed, the function will return false (0).
The Click function is a level sensitive function, which means that the mouse button(s) must be pressed when the function is executed or it will return false. This means that a brief press of a mouse button with the cursor in the correct target area might not be picked up by the Click function if the system is heavily loaded. Use the Pick() function for action triggers, rather than Click().

Note: If Click is used as an action trigger it will not reset; the action will continue to trigger as fast as possible as long as the Click is true. This is different from Pick, which would trigger once and be reset by the action trigger.

Example:

```
Box(100, 500, 500, 100 { Coordinates for the box },
     1 { Solid style },
     5 { Width in screen coordinates },
     Cond(Click(100, 500, 500, 100, 56 { Color depends on click }),
          12 { Color is red when clicked     },
          7  { Color is light gray otherwise })
    );
```

This draws a blue box on the screen. If the mouse button is inside the square and any button combination is held down, the box will turn white. If the mouse should move outside the box, or if all buttons are released, the Box color will turn back to gray. Note that the Box coordinates and the coordinates for the Click function are the same by choice - they are not required to match. That is, the coordinates in the Box specify where to draw the box, and the coordinates in the Click specify where to look for a mouse click.

ClientSocket

Description: Opens a client WinSock-compliant socket stream and returns a stream value, or a numeric error code.

Returns: Varies – see comments

Usage: Script Only.

Function Groups: Stream and Socket

Related to: CloseStream | ServerSocket | SocketAttribs | SocketServerEnd | SocketServerStart | SocketWait | SRead | SWrite | TCPIPReset

Format: ClientSocket(Protocol, Host, Service, TransmitLen, ReceiveLen, Flush[,ProtocolFilters, InboundPortOrStream, IPOut, PortOut)

Parameters:

Protocol

Required. Any numeric expression giving the protocol to be used. This must be a valid 0 for TCP/IP protocol or a valid 1 for a UDP protocol.

Host

Required. Any text expression giving the host name or TCP/IP address to connect.

Service

Required. Either any text expression giving the service name to connect, or any numeric expression giving the port number with which to connect.

TransmitLen

Any numeric expression for the number of bytes to buffer when transmitting. The value must be a signed long integer, where only positive values are useful.
If the application is running on a operating system of Windows 7 / Server 2008 R2, or later, and the value is set to zero, then Windows will manage the appropriate buffer size for the link speed and latency.
If you set the buffer size, the value should match or be larger than the largest message that is expected.

A high bandwidth / high latency link will require a larger size in order to achieve optimum efficiency, but the exact size can be determined only by empirical testing.

ReceiveLen

Required. Any numeric expression for the maximum number of bytes to buffer by VTScada when receiving. Additional buffering will be handled by WinSock.

The value must be a signed long integer, where only positive values are useful.

If the application is running on a operating system of Windows 7 / Server 2008 R2, or later, and the value is set to zero, then Windows will manage the appropriate buffer size for the link speed and latency.

If you set the buffer size, the value should match or be larger than the largest message that is expected.

A high bandwidth / high latency link will require a larger size in order to achieve optimum efficiency, but the exact size can be determined only by empirical testing.

Flush

Required. Any logical expression. If true, the transmit buffer will be flushed (transmitted) after each write to the stream.

This normally should be false to reduce network traffic by allowing the driver to group smaller packets into a single larger packet.

ProtocolFilters

Optional. A 2-dimensional array rrepresenting a stack of engine protocol filters and their parameters. These filters are used to process or modify data before transmission or after reception. No default is provided.

Example:

```
PFilter = New(2);
PFilter[0] = New(2);
PFilter[0][0] = "SSL";
PFilter[1] = New(2);
PFilter[1][0] = "NULL";
PFilter[1][1] = "";
```

InboundPortOrStream

Optional. Used only in connection with a UDP connection.

If set, this should be an existing UDP stream as returned from a ServerSocket, for the same remote IP as is being connected to.

Incoming UDP datagrams on the ServerSocket stream will continue to be received, but the stream can also be used for datagram transmission. If InboundPortOrStream is not a stream, it is interpreted as a local port number on which to listen for inbound datagrams.

The stream returned by ClientSocket can be used, in both cases, for datagram transmission and reception.

IPOut

Optional Used to identify the IP of the network interface card from which to transmit UDP datagrams. LocalIP s of use on multi-homed machines to identify the physical IP binding to use. No default is provided.

PortOut

Optional. Used to identify the local port from which to transmit UDP datagrams No default is provided.

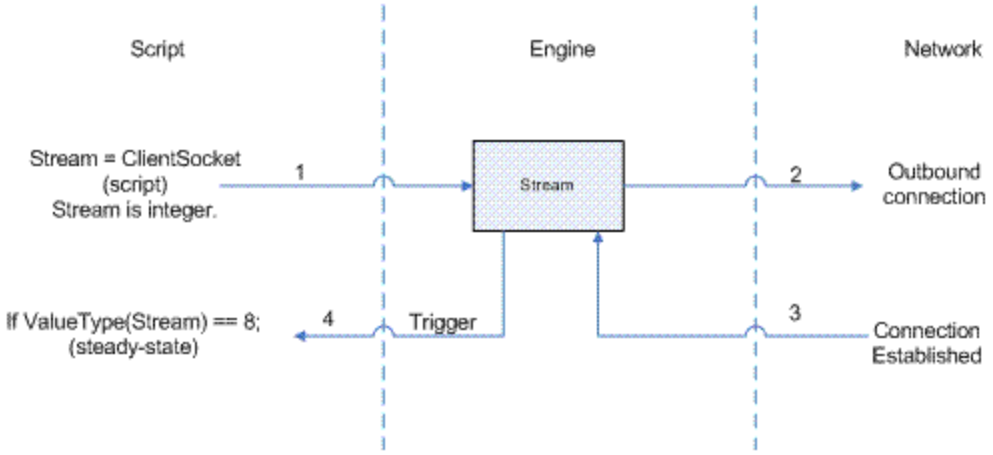
Comments This function will return its (integer) socket number when the socket is created but not yet connected, a stream value when the connection is made, or a short integer error code. If the socket connection is lost (server shutdown) the stream is closed and set invalid (no error code returned).

Error Code	Description

10061	Cannot connect to Host at port number specified
11004	Cannot find requested host

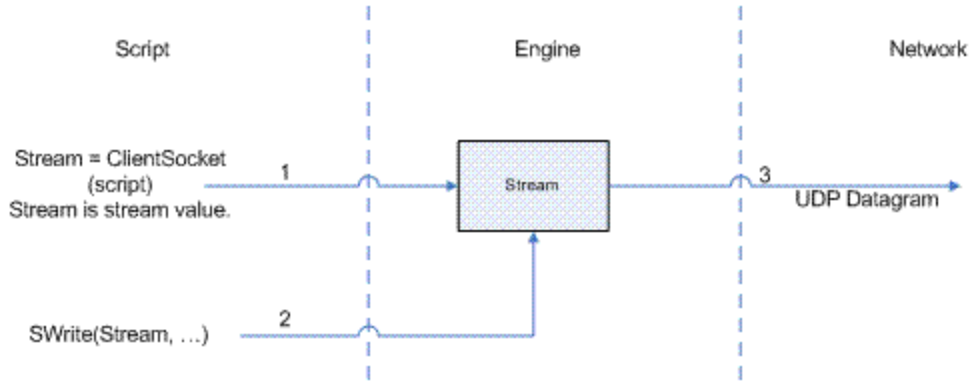
The client socket function has a slightly different behavior, depending on whether the connection is made via TCP or UDP. The difference is explained in the following two diagrams.

Client Sockets on TCP:



1. A ClientSocket statement runs and returns an integer value.
2. An outbound connection is made.
3. When the connection is established, the stream is triggered.
4. The stream trigger causes the value returned from 1 to become a stream value.
   - If the connection attempt fails at any point, the value returned from 1 will become a negative integer, representing an error code

Client Sockets on UDP:



1. A ClientSocket statement runs and returns a stream value.
2. Stream writing statements are used to write data to the stream
3. A UDP datagram is issued to the target device each time a write is done by script code.

Example:

```
Init [
  If 1 Main;
  [
    client = ClientSocket(0 { TCP/IP protocol },
                          "WServer" { Host },
                           20000 { Port number },
                           1024, 1024 { Buffering },
                           1 { Flush after writes });
  ]
]
Main [
  { If stream connection lost, retry connection }
  If TimeOut(! Valid(client), 2) Init;
  { Exit if return value valid and not a stream }
  If ValueType(client) != 8 Error;
  { Read stream data as received or on demand with "r" key }
  If GetStreamLength(client) > 0 || MatchKeys(2, "r");
  [
    SRead(client, Concat("%", Concat(GetStreamLength(client),
         "c")), data);
  ]

  { Write stream data to server every second }
  If TimeOut(1, 1);
  [
    SWrite(client, "%s", Concat(" Hello World ",
          Time(Seconds(), 3)));
  ]
  { Close stream if window closed, then stop }
  If WindowClose(Self());
  [
```

```
    CloseStream(client);
    Slay(Self(), 1) ;
  ]
  { Display received data and connection status }
  ZText(10, 150, data, 2, 0);
  ZText(200, 100, Cond(ValueType(client) == 8,
  "Connected", "Not Connected"), 10, 0);
]
Error [
  { Display error code }
  ZText(100, 130, Concat("Client error code : ", client),
       10, 0);
]
```

ClipboardGet

Description: Returns the current contents of the system clipboard as a string. This function enables an application to perform text "paste" operations.

Returns: Text

Usage: Script Only.

Function Groups: Clipboard, String and Buffer

Related to: ClipboardPut

Format: ClipboardGet()

Parameters: None

Comments: This function can be used to obtain the current contents of the clipboard as a string. If the clipboard does not contain a textual value, then the function will return invalid.

ClipboardPut

Description: Set the current contents of the system clipboard to a string. This function enables an application to perform text "copy" or "cut" operations.

Returns: Nothing

Usage: Script Only.

Function Groups: Clipboard, String and Buffer

Related to: ClipboardGet

Format: ClipboardPut(String)

Parameters:

String

Required. Any text expression that will be copied to the system clipboard.

Comments: This function can be used to set the current contents of the clipboard. If the parameter is not a text expression, then the contents of the clipboard are left unchanged; otherwise the current contents of the clipboard are overwritten.

CloseStream

Description: Closes and flushes any type of open stream and returns its own error code.

Returns: Boolean

Usage: Script Only.

Function Groups: Stream and Socket

Related to: BuffStream | FileStream | PipeStream | StreamEnd

Format: CloseStream(Stream)

Parameters:

Stream

Required. Any expression that returns a stream value.

Comments: The return value is true if the stream were successfully closed; invalid otherwise.This function closes the stream passed to it.
If the stream is invalid, or the stream is already closed, nothing happens.

Example:

```
If GetStreamLength(stream) == 0 && ! Valid(closeOK);
[
  closeOK = CloseStream(stream);
]
```

This example checks to see if the stream has any data in it, and if it is empty and has not already been closed, closes it, setting closeOK to 1 if the stream were closed and invalid otherwise.

Cls

Description: Clears the screen and sets its background color.

Returns: Nothing

Usage: Script or steady state.

Function Groups: Color, Graphics, Window

Related to: GetSystemColor

Format: Cls(Color)

Parameters:

Color

Required. Any numeric expression giving the color to which the screen should be set.

You may use any of the following methods to specify the color:

- a palette index VTScada Color Palette
- a system color Constants for System Colors
- an RGB string with optional Alpha value in the format, "<AARRGGBB>", or "<RRGGBB>", where AA, RR, GG and BB are hexadecimal digits.

Note that Cls will ignore the alpha channel if you provide the color as an <AARRGGBB> string, since the background color of a window cannot have an alpha value that is different from the window itself.

Comments: This statement sets the background color for the

window that the function is executed within. It will clear the screen but any layered graphics such as GUI or Z-graphic functions will be redrawn.
The description of the color parameter mentions that Cls will ignore alpha (transparency) values.
Note also that you cannot set the background color of windows that use their background as a transparency mask (bit 18 set in the Style parameter).
Cls will work on both workstations and on internet clients.

Example:

```
Cls(1);
```

This sets the window background color to dark blue.

CodeText

Description: Returns information (usually the source code text), matching the given code value.

Warning: For use by advanced programmers only.

Returns: Varies (see 2nd parameter).

Usage: Script Only.

Function Groups: Advanced Module

Related to: SetCodeText

Format: CodeText(CodeVal[, Type])

Parameters:

CodeVal

Required. Any code value giving the statement or function within a statement whose text to be read.

Type

Optional numeric expression. Controls what will be returned by the function according to the following table:

Type	Description
0	default. Return the text for the expression.
1	Return the offset in the file for the start of the expression.
2	Return the size (number of bytes) of the expression.

Comments: Returns the source code text, offset or size for the provided code value. Typically followed by a SetCodeText call, which would then update the source code with new text.

ColorSelect

(System Library)

Description: Color Selection Tool. This module draws a color selection button and its accompanying display area.

Returns: Nothing

Usage: Steady State only.

Function Groups: Color, Graphics

Related to:

Format: \System\ColorSelect(X1, Y1, X2, Y2, Color [, BtnLabel, BtnOnLeft, Standard, VertAlign, FocusID, OpenDialog, NoTrans, BGColor, FGColor, ShowIndex])

Parameters:

X1

Required. Any numeric expression giving the X coordinate on the screen of one side of the button or

color display area. The smaller of values X1 and X2 will always be the left side.

Y1

Any numeric expression giving the Y coordinate on the screen of either the top or bottom of the button and color display area. The smaller of values Y1 and Y2 will always be the top.

X2

Required. Any numeric expression giving the X coordinate on the screen of the side of the button or color display area opposite to X1.

Y2

Required. Any numeric expression giving the Y coordinate on the screen of the top or bottom of the button and color display area, whichever is the opposite to Y1.

Color

Required. The variable whose value is set to the index of the chosen colore.

BtnLabel

Optional. Any text expression to be used as a label on the color selection button. No default if missing or Invalid.

BtnOnLeft

Optional Any logical expression. If true (non-0) the button will appear to the left of the color display area, If false (0) it will be to the right. The default value is true.

Standard

Optional Any logical expression, If true (non-0) the button and color display area will be standard system size. If false (0) they will be sized to fit their boundaries

and VertAlign will be ignored. The default value is true.

VertAlign

Optional Any numeric expression that sets the vertical alignment of the button and display area according to one of the following options:

Value	Vertical Alignment
0	Top
1	Center
2	Bottom

If Standard is true, this parameter is ignored. The default value is 0, top alignment.

FocusID

Optional Any numeric expression for the focus number of the button. If this value is 0, the current value of Color will still be displayed, but it will not be able to be set because the button will appear grayed out. The default value is 1.

OpenDialog

Optional Any logical expression. If true (non-0) the dialog will be open. If false (0), it will be closed. The default is true.

NoTrans

Optional Enables you to turn off the transparent color option. If NoTrans is set to true (1), then the transparent color option will be turned off.
If NoTrans is set to false (0), then the transparent color option will be turned on. (The transparent color option is set within the color selection dialog using the "Transparent" (or "Transparent Brush" or "Transparent Pen") check box.)
No default value is provided.

BGColor

Optional. Any numeric expression for the background color of the control. No default value.

FGColor

Placeholder for the foreground color of the control. Not currently implemented.

ShowIndex

Optional. Color index is shown when TRUE. Defaults to TRUE if not specified.

Comments:

This module is a member of the System Library, and must therefore be prefaced by \System\, as shown in the "Format" section. If you are developing a script application, use "System\..." rather than "\System\..." in the function call.

If the button and display area are set to be standard size, the button size will remain a constant size (101 x 31 pixels) regardless of the boundaries of the defined area, but the display area will vary between a minimum (square) size to a maximum length equal to the length of the button, depending on the size of the bounding area. Once the maximum display area size has been reached, the button and area will be fully justified to cover the entire width of the defined area (the vertical alignment will be set by the value of VertAlign).

For any optional parameter that is to be set, all optional parameters preceding the desired one must be present, although they may be invalid.

Examples:

```
System\ColorSelect(10, 300, 210, 200 { Location of btn/display },
MyColor { Variable storing color },
         "Set Pipe Color" { Button label   },
         1                { Button on left },
         1              { Standard sizing },
```

```
1               { Center btn/display },
3               { Focus ID of button });
```

Combine

Description: Performs a Merge2 operation with automated conflict resolution and change priority.

Returns: Merge buffer

Usage: Script Only. (always called as a subroutine)

Function Groups: Configuration Management

Related to: Diff | Merge2 | Merge

Format: \LayerRoot\Combine(Source, Diff1, Diff2, SrcPath, pFail)

Parameters:

Source

Required. The buffer or stream to be modified

Diff1

Required. High priority diff to merge to Source.

Diff2

Required. Low priority diff to merge to Source.

SRCPath

Required. Full file path for the origin of the Source buffer.

pFail

Required. Pointer to a variable in which an error message will be returned.

Comments: This module must be called as a subroutine.
This function is guaranteed to return a result, however it will discard changes if unable to merge them without conflict. It is the recommended method for combining different change paths against standard VTScada file types (settings files, source files, tag

files, etc.).
Changes from the high priority diff are treated preferentially to changes from the low priority diff when resolving conflicts. If the operation fails then the result will be the original buffer modified by the high priority diff only. Failure is indicated by the pFail parameter being set to a value other than zero; Typically a string detailing the type of failure.
Notes:

- All Tag file conflicts are resolvable, Combine does not fails on tags.
- Conflicts of Adjacency (where two change regions share an edge but do not overlap) are resolved for all non-tag files.
- Simultaneous Addition Conflicts (where two pure addition changes coincide) are resolved for Page, Menu, Setting, and PageNote files.
- Specific segments of AppRoot files are parsed and rebuilt allowing for all types of conflicts to be resolved, these are additionally checked for duplicate entries, which are removed.

The results of conflict resolution are passed to the Mend function in order to generate complete merged file buffers. This module returns a simple merge of the high priority diff and the source buffer should conflicts occur and conflict resolution fail. In this case *pFail is set to an error string that describes the failure in a human-readable form.

COMClient

Description: Instantiates COM objects that do not possess a user interface.

Returns: COM client interface

Usage: Script or steady state.

Function Groups: COM

Related to: ActiveX | ActiveX | COMEvent | COMStatus | Caller

Format: COMClient(ObjectIdentifier [, ObjectContext, EventSearchScope, EventParent, EventCaller])

Parameters:

ObjectIdentifier

Required. Specifies a unique identifier for the object to be instantiated. It may take one of the following forms:

- a text string representing a ProgID (e.g. "Excel.Application").
- a textual GUID, in registry format (e.g. "{00020812-0000-0000-C000-000000000046}").
- a binary GUID (e.g. the result from "GetGUID(1, 00020812-0000-0000-C000-000000000046)").

ObjectContext

Optional. If present, this object specifies the contexts in which it is permissible to instantiate the COM object.
A subset of values taken from the CLSCTX enumeration is supported. Possible values are:

- CLSCTX_INPROC_SERVER (1): The COM object is instantiated in the VTScada process (i.e. by a DLL).
- CLSCTX_LOCAL_SERVER (4): The COM object is instantiated in a separate process (i.e. by an .exe, but only on the same machine upon which VTScada is running.

- CLSCTX_SERVER (13): Any of the above. This is the default and permits the COM object to be instantiated wherever the "class factory" (which performs object instantiation) sees fit.

EventSearchScope

Optional May be any expression that yields a module value or object value. If present, this parameter specifies the scope in which to search for event subroutines. No default value is provided.

EventParent

Optional. May be any expression that yields an object value. If present, specifies the context that is used to resolve scope for event subroutines.
If absent or Invalid, defaults to Self().

EventCaller

Optional. May be any expression that yields an object value. If present, specifies an "auxiliary" context for event subroutines. An event subroutine can retrieve this value using Caller(Self()).
If absent or Invalid, defaults to Self().

Comments:

COMClient instantiates a COM object. Generally this statement is used to instantiate COM objects that do not possess a user interface. If the object does display a user interface, then the user interface will appear in a window created and owned by the object. It is more usual to use the ActiveX statement to create a COM object that has a user interface.
If the statement succeeds, a COM client interface is returned, allowing subsequent access to the object. If the statement fails, Invalid is returned.

EventSearchScope specifies the location where the event subroutines for this COM object instance may be found. Each event subroutine is named after the corresponding event produced by the default outgoing interface for the COM object. If the COM object generates an event for which an event subroutine cannot be found, the COM object is informed that event handling for that event is not implemented.

When an event subroutine is run in response to an incoming event from the object, the parent and caller for the subroutine are as specified by EventParent and EventCaller. The event subroutine may update any of its parameters, and any that are defined as [in, out] or [out]; these will be returned to the COM object on completion of the event subroutine. The event subroutine should return an integer value, which is returned to the COM object as the result of the event. A value of 0 indicates the event completed successfully.

If the COM object to be instantiated is an inproc server object (i.e. it is a DLL) then it must be a 64-bit COM server for 64-bit VTScada and a 32-bit COM server for 32-bit VTScada. If it is instantiated out-of-process (i.e. it is an .EXE) then either 32-bit VTScada or 64-bit VTScada can work with either 32-bit or 64-bit COM servers.

If used in a script, the COM object will remain instantiated until the last reference to that object has been invalidated. You assign the return value of the COMClient script statement to a variable, and the COM object will remain instantiated as long as that variable or any other variable holds a copy of the COM Client Interface handle. Only when the last copy of the COM Client Interface handle has been destroyed will the COM object be destroyed.

If used in a steady-state statement, the COM object will only remain instantiated while the steady-state statement is still running; in other words, a change of state or destruc-

tion of the module instance that is running the statement will cause the COM object to be destroyed. Any variables that hold a handle to the COM Client Interface will be invalidated at that time.

COMEvent

Description: Sets an event subroutine context for an existing COM client interface.

Returns: Text

Usage: Script Only.

Function Groups: COM

Related to: ActiveX | COMClient | COMEvent | COMStatus

Format: COMEvent(COMClientInterface [, EventSearchScope, EventParent, EventCaller])

Parameters:

COMClientInterface

Required. A COM client interface handle returned from a COMClient or ActiveX statement.

EventSearchScope

Optional May be any expression that yields a Module or an object value. If present, specifies the scope in which to search for event subroutines. No default value is provided.

EventParent

Optional May be any expression that yields an object value. If present, specifies the context that is used to resolve scope for event subroutines. If absent or Invalid, defaults to Self().

EventCaller

Optional May be any expression that yields an object value. If present, specifies an "auxiliary" context for

event subroutines. An event subroutine can retrieve this value using Caller(Self()). If absent or Invalid, defaults to Self().

Comments: The COM client interface specified as the first parameter may or may not already have an event subroutine context associated with it. This function supplies a new context that destructively replaces any existing context. Use of this function enables dynamic modification of the event subroutine context. The context may also be dynamically set if the COMClient or ActiveX function is being run in steady state, by simply changing the parameters of the steady-state statement. It is more usual to use this function for COM client interfaces that are created by a script statement.

EventSearchScope specifies the location where the event subroutines for this COM object instance may be found. When an event subroutine is run in response to an incoming event from the object, the parent and caller for the subroutine are as specified by EventParent and EventCaller.

CommaFormat

(System Library)

Description: Returns a number as text with embedded commas.

Returns: Text

Usage: Script or steady state.

Function Groups: Math - Generic Functions, String and Buffer

Related to: FormatNumber | Format

Format: \System\CommaFormat(Color)

Parameters:

Value

Required. Any integer expression giving the value to be formatted.

Comments: Note that this function is part of the System library and must be preceded by \System\. This function will not format floating point numbers – the decimal point will be treated as a digit.

Example:

```
\System\CommaFormat(123456);
```

Returns "123,456".

CommandLine

Description: Returns any command line arguments as a text string.

Returns: Text

Usage: Script Only.

Function Groups: Software and Hardware

Related to: Version | SerialNum

Format: CommandLine()

Parameters: None

Comments: None

Example:

If VTScada was started from the Windows Program Manager "Run" option with:

```
VTS5 User423 Stn62
```

then any application with the following line:

```
Text(10, 10, CommandLine(), 12, 0);
```

would print out in the upper left hand corner of the window:

```
User423 Stn62
```

Commission

(Alarm Manager module)

Description: Commission the alarm by adding it to the Configured list, or modify an existing alarm's configuration.

Returns: Nothing

Usage: Script Only.

Function Groups: Alarm

Related to: Decommission | GetAlarmConfiguration | EvaluateAlarm

Format: \AlarmManager\Commission(AlarmObj, CfgStructure[, Value, ValueTimestamp, SuppressConfigEvent, AlarmDB, MachineID)

Parameters:

AlarmObj

Required. The alarm object. Normally, the unique ID of the tag within which the alarm is being commissioned.

CfgStructure

Required. A structure of alarm configuration parameters. Normally obtained by a call to \AlarmManager\GetAlarmConfiguration.

Value

Optional numeric. Current value to evaluate, based on the new configuration.

ValueTimestamp

Optional UTC timestamp of the value. Defaults to the current time.

SuppressConfigEvent

Optional Boolean. If TRUE, no transaction record will be stored for this call to Commission. Should be used for alarm parameters that may be updated often due to being set by tag values or expressions rather than con-

stants. Defaults to FALSE.

AlarmDB

Optional name or object. The alarm database to be used for this alarm. Not necessary if the alarm object is valid.

MachineID

Optional. The workstation ID to be associated with the alarm. Defaults to the current workstation.

Comments:

Commission should be used when creating an alarm, or when updating that alarm's configuration. It will always generate a new call to EvaluateAlarm with the given value of the trigger, evaluating that against the current setpoint and comparison function.

Commission will store the alarm's object and its Root value in a dictionary of alarm names. This gives an efficient look-up table to get an alarm object.

After an alarm has been commissioned, further calls to this function will update that record in the alarm database. A call to \AlarmManager\Evaluate is made automatically as part of each call to Commission.

Note: For the sake of creating efficient code, commission should never be used as a substitute for EvaluateAlarm. Use commission only when creating or changing alarm configuration, not when handing new values to the alarm to be evaluated against the setpoint.

Example:

The following would typically be found in a tag's Refresh state. This alarm will activate when the tag's value becomes 1.

```
IfElse(Valid(Name), Execute( { create or obtain the configuration
structure for this alarm }
                             Cfg          = \AlarmMan-
ager\GetAlarmConfiguration(UniqueID);
                               { update the property values in that
structure }
                             Cfg\Name     = UniqueID;
                             Cfg\Area     = Area;
                             Cfg\Priority = PriorityValue;
                             Cfg\Setpoint = 1;
                             Cfg\Function = \AlarmManager\ALM_FUNC_
EQUAL;
                             { ... other configuration properties as
required ... }
                               { commission, or update the commission
of, the alarm }
                             \AlarmManager\Commission(Root, Cfg,
Value);
                          );
);
```

Related Information:

See: "Alarm API Structure Definitions" in the VTScada Programmer's Guide for Alarm configuration structures

CommitEditedFiles

Description: This function compiles and commits edited files if the compile succeeds.

Returns: Object (which becomes Invalid upon completion)

Usage: Script Only.

Function Groups: Configuration Management

Related to: EditFile | DirectApply

Format: LayerRoot\CommitEditedFiles(User, Comment, AlreadyHasLock[, ReloadOnFailure, pFail, RSema, SuppressAudit])

Parameters:

User

Required. The user performing work on the repository's working copy.

Comment

Required. Text that will be stores with the repository commit.

AlreadyHasLock

Required. Boolean indicating whether we already have a working copy lock.

ReloadOnFailure

Optional Boolean. If set TRUE, then if the files cannot be integrated into a running system, the files are reverted and no reloading of the reverted files is performed. If set FALSE, then the reverted files are reloaded.
This can be set to false when you are sure that none of the changes being committed have been integrated into a running application.
Defaults to TRUE.

pFail

Optional pointer to a Boolean. Set TRUE upon failure.

RSema

Optional repository semaphore. If provided, it is essential that you also have the working copy lock.

SuppressAudit

Optional Boolean. Set TRUE to suppress the audit the commit. This is typically done because you are guaranteed to do another commit before you give up the WC lock.
Defaults to FALSE.

Comments: This helper function manages the Working Copy files by applying a set of file changes indicated in prior EditFile calls.
Typically the caller will have the working copy lock and this call is made using only the first three parameters, with the third parameter set TRUE. This operation returns an object that becomes invalid

upon completion.
If the compile does not succeed, this function will revert the compilation changes. By the time this launched module terminates, the Modified dictionary is always emptied.
This module launches a worker module into the Layer so that the operation is not interrupted by this module's caller being slain.

Compile

Description: Compiles text and creates a new function; its type of return value is determined by its input parameters.

Warning: This function may cause irrecoverable alteration of your application. It should be used only by advanced programmers.

Returns: Varies

Usage: Script Only.

Function Groups: Compilation and On-Line Modifications

Related to:

Format: Compile(FuncNames, OpCodes, Number, Module, Script, Sense, Stream, ClassBuffer, NumClass, TStateBuffer, TActionBuffer, NumTokens, CStateBuffer, CActionBuffer, CDataBuffer, Error, ParserStack, LineCount, Column, Count)

Parameters:

FuncNames

Required. Any array expression for the function names list.

OpCodes

Required. Any array expression for the Operational Codes that correspond to the names in FuncNames.

Number

Required. Any numeric expression for the number of FuncNames/OpCodes pairs.

Module

Required. Any expression that returns a module value. This is the context where the compile takes place.

Script

Required. Any logical expression. If true, the text will be compiled as a script statement. If false, it will be compiled as a normal statement.

Sense

Required. Any logical expression. If true, this statement will be compiled as case-sensitive, otherwise, this will be case-insensitive.

Stream

Required. Any expression for the input stream to read.

ClassBuffer

Required. Any text expression that provides the tokenizer look-up table.

NumClass

Required. Any numeric expression for the number of character classes in the tokenizer state table.

TStateBuffer

Required. Any expression for the tokenizer state table. The tokenizer begins in state 0.

TActionBuffer

Required. Any two-dimensional array expression for the tokenizer action table.

NumTokens

Required. Any numeric expression for the number of tokens in the compiler table.

CStateBuffer

Required. Any text expression for the compiler state table. The compiler begins in state 0.

CActionBuffer

Required. Any text expression for the compiler action table.

CDataBuffer

Required. Any two-dimensional array expression for the compiler data table.

Error

Required. Must be a variable into which the resulting error code will be stored. The error code may be any of the following:

Error	Description
0	No error.
-1	Operation code in OpCode table is not defined.
-2	Parameter must be a constant.
-3	Too many parameters.
-4	Compiler table index past the end of the table.
-5	Token type is < 0 or > max_token.
-6	illegal attempt to add op code from built-in function.
-7	Op code array entry is invalid.
-8	Illegal attempt to add variable parameter.
-9	Too many parameters.

-10 Illegal radix in compiler data table.

-11 Illegal digit in integer.

-12 Illegal action in CActionBuffer.

-13 No function specified.

-14 Not enough parameters.

-15 Does not have a single parameter.

-16 Compiler stack is too deep (> 1000)

-17 Infinite loop detected.

-18 Suspending and exiting at the same time.

ParserStack

Required. Any stack used to govern the compile.

LineCount

Required. Must be a variable. The current line (number of carriage returns) will be stored here.

Column

Required. Must be a variable. The current column will be stored here.

Count

Required. Must be a variable. The number of characters processed is stored here.

Comments: This function is used when compiling text statements. The syntax of the statements is described by the table parameters that are not fully described here. This function compiles text to create a new function, but doesn't add the new function.

COMPort

Description: Opens a serial port and handles all interrupts and asyn-

chronous events for that serial port, including transmission, reception, and control. It returns its own error code.
Please note that the SerialStream function is generally preferred in many situations; however, ComPort continues to be supported.

Returns: Numeric

Usage: Steady State only.

Function Groups: Serial Port

Related to: ActiveX | SerCheck | SerialStream | SerIn | StrLen | SerOut | SerRcv | SerRTS | SerSend | SerString | SerStrEsc | SerWait

Format: ComPort(Port, ReceiveLen, TransmitLen, Baud, DataBits, StopBits, Parity, RTS, XOnXOff, Obsolete, Obsolete, Control,0,0,0,0,0,0,0,0,0)

Parameters:

Port

Required. Any numeric expression giving the serial port number to be used.
For COM1, Port = 1;
For COM2, Port = 2.
The valid range for Port is 1 to MaxComPorts (a variable storing the maximum number of Windows serial ports available. As of the release date of VTS 9.1, this is 4096).

ReceiveLen

Required. Any numeric expression giving the size of the receive buffer in bytes. ReceiveLen must be in the range 2 to 32 766.
If more bytes are received than can fit in the receive buffer before your application removes them using SerRcv or a similar WEB function, the additional data will

be lost.

TransmitLen

Required. Any numeric expression giving the size of the transmit buffer in bytes. TransmitLen must be in the range 2 to 32 766.
The buffer must be large enough to hold the maximum number of bytes pending transmission at any instance.

Baud

Required. Any numeric expression giving the baud rate. Baud must be in the range 10 to 115 200, and must divide evenly into 115 200 with no more than 2.5% error.

DataBits

Required. Any numeric expression giving the number of data bits per character. DataBits must be 5, 6, 7, or 8.

StopBits

Required. Any numeric expression giving the number of stop bits per character. StopBits must be 1 or 2.

Parity

Required. Any numeric expression giving the parity checking to use (as follows)

Value	Parity
0	No parity
1	Odd parity
2	Even parity
3	0 Stick (space parity)
4	1 Stick (mark parity)

RTS

Any numeric expression that gives the RTS buffer control method.

RTS is on while transmitting. When a transmission is complete, RTS is off. This is usually used to control the transmitters on RS-422/485 ports.

This parameter has no effect if the automatic RTS control is selected in the Control parameter.

Acceptable values of the RTS parameter are as follows:

Value	RTS Method
0	Force RTS off
1	Force RTS on
2	Half-duplex operation (Windows NT only)
3	Controlled by SerRTS function

If this parameter is 2, the SerRTS function can set its value, however, regardless of SerRTS, the RTS control line will be asserted when data is sent.

If the SerRTS is called to change the RTS line while data is being transmitted, the RTS line will not change when the last byte is sent.

If SerRTS is not executed while the data is transmitted, the RTS line will be cleared after the last byte is transmitted.

XOnXOff

Required. Any logical expression. If true (non-0), software flow control is to be used. If false (0), flow control software is not used.

Obsolete

No longer used, but is maintained for backward compatibility with previous versions of VTScada. Set to 0.

Obsolete

No longer used, but is maintained for backward compatibility with previous versions of VTScada. Set to 0.

Control

Any numeric expression that specifies the handling procedure for the clear to send (CTS), carrier detect (CD), and data set ready (DSR) input lines, and the data terminal ready (DTR) output line on the serial port. The value must be in the range 0 to 63. The desired action is the sum of the following values:

Value	Bit No.	Control
1	0	DTR on (otherwise DTR is off)
2	1	Enable CTS control
4	2	Enable CD control
8	3	Enable DSR control
16	4	Enable RTS/CTS control
32	5	Enable DTR/DSR control

- If bit 1, CTS control, is set data will only be transmitted if the CTS signal is on. If CTS control is disabled, the CTS line is ignored.
- If bit 2, CD control, is set data will only be transmitted when the CD signal is on. If CD control is disabled, the CD line is ignored.
- If bit 3, DSR control, is set data will only be transmitted when the DSR signal is on. If DSR control is disabled, the DSR line is ignored.
- If bit 4, RTS/CTS control, is set the CTS control behaves as described above, and the RTS line will be held high until the receive buffer reaches 75% full. It will then go low, indicating to the other device to stop transmitting data. The RTS line will

go high again when the receive data buffer drops below 25% full. This is known as hardware flow control. RTS/CTS control enabled overrides the RTS parameter.

- If bit 5, DTR/DSR control, is set the DSR control behaves as described above, and the DTR line will be held high until the receive buffer reaches 75% full. It will then go low, indicating to the other device to stop transmitting data. The DTR line will go high again when the receive data buffer drops below 25% full. This is known as hardware flow control. DTR/DSR control enabled overrides bit 0, DTR on option.
- bits 6 through 14: Obsolete. No longer used, but maintained for backward compatibility with previous versions of VTScada. Set to 0.

Obsolete x 9

Required. Nine parameters following Control are obsolete, but must be included for the function to compile. Normally set to 0's.

Comments: This statement has been superseded by the SerialStream function and is maintained for backwards compatibility only.
This function is part of the driver toolkit and must be active

Value	Meaning
0	No error
1	Out-of-range error in one of the parameters
2	Port already in use or not available

The return value is an error code having one of the following meanings:

Value	Meaning
0	No error
1	Out-of-range error in one of the parameters
2	Port already in use or not available
5	Access denied
31	General failure
87	Invalid parameter

A ComPort function must be active for serial port communications. None of the driver toolkit functions (anything beginning with Ser... , such as SerSend) will work without a ComPort function.

Make sure that VTScada's mouse (if it is serial) is on a different port, because the mouse and ComPort can interfere. Also make sure that no other hardware or software is interfering with the serial port hardware interrupts (IRQ4 for COM1:, IRQ3 for COM2:). Network cards often use IRQ4, which will cause a problem with a mouse or ComPort on COM1.

Before writing a communications driver for VTScada, it is important to understand some general data communications concepts (such as headers, checksums, packets, etc.) as well as the particular protocol you wish to use. It is a good idea to understand how fixed modules may be used to provide semaphores and queues (so that modules designed to read packets can queue up for their turns to use the serial port, and prevent collisions).

Example:

```
ComPort(2 { COM2: },
        1024 { Buffer 1024 bytes of received data },
        1024 { Buffer 1024 bytes of transmitted data },
        9600 { Baud rate },
        8 { Data bits per byte },
        1 { Stop bit per byte },
        0 { No parity bit },
        1 { Force RTS on },
        0, 0, 0 { Obsolete parameters },
        3 { Control: DTR On, CTS control enabled },
        0, 0, 0, 0, 0, 0, 0, 0, 0 { Obsolete parameters });
```

This example opens COM2: for use with serial port functions. These functions should use 2 as their Port parameter.

Compress

Description: Eliminate invalid array entries.

Returns: Nothing

Usage: Script Only.

Function Groups: Array

Related to: ArrayOp1 | ArrayOp2 | AValid | Filter

Purpose: This statement moves all of the valid array entries to the first part of the array, maintaining the order of the valid elements.

Format: Compress(ArrayElem, N)

Parameters:

ArrayElem

Required. Any array element giving the starting point for the array search. The subscript for the array may be any numeric expression.If processing a multidimensional array, the usual rules apply to decide which dimension should be examined.

N

Required. Any numeric expression giving the number

of array elements to use starting at the element given by the first parameter.
If N extends past the upper bound of the array dimension, this computation will "wrap-around" and resume at element 0, until N elements have been processed.

Comments: None

Example:

```
Init [
  If 1 Main;
  [
    x[0] = 4;
    x[1] = Invalid;
    x[2] = 5;
    x[3] = 6;
    x[4] = Invalid;
    x[5] = 3;
    Compress(x[0], 6);
  ]
]
```

This will result in the first 4 elements of array x being set to 4, 5, 6 and 3 respectively, with the last 2 elements being set invalid.

COMStatus

Description Returns the last status information that occurred for a specified COM client interface.

Returns Varies

Usage Script Only.

Function Groups COM

Related to: ActiveX | COMClient | COMEvent

Format ComStatus(COMClientInterface [, Options])

Parameters

COMClientInterface

Required. A COM client interface handle returned from a COMClient or ActiveX statement.

Options

Optional. Any numeric expression. Set to 0 to return the Windows API error code.
Set to 1 to return a text message representation of the Windows API error code.
Defaults to 0 if missing or Invalid.

Comments The value returned indicates the success or failure of the last operation attempted through the specified COM client interface.
Depending on the Options parameter, this function returns either the Windows API error code or a text value containing an ASCII representation of the error code. There is no guarantee that all error codes can successfully be translated into a text representation.

Concat

Description: Returns the text value that is the concatenation of all the text parameters.

Returns: Text

Usage: Script or steady state.

Function Groups: String and Buffer

Related to: StrCmp | StrICmp | StrLen | SubStr

Format: Concat(String1, String2[, String3, ...])

Parameters:

String1, String2, String3

A minimum of two parameters are required May be any expressions that give the text values to add to the resulting string.

Comments: This function takes all of the characters of the first parameter and appends all of the characters of the second parameter to the end of this string. It will then take all of the

characters of the next parameter and append them to the result. This will continue until there are no more parameters.
Since the text strings are delimited by quotation marks, to include a set of quotation marks as part of the resulting string, you must use two sets of quotation marks (see example).

Example:

```
ZText(10, 10, Concat("The maximum is ", maxValue,
      " and the minimum is ", minValue), 2, 0);
```

This statement displays the 2 strings and 2 values as a single line of (white) text in the window.

```
ZText(10, 10, Concat("""Hello"", said the ", animal), 12, 0);
ZText(10, 10, Concat("""", "Hello", """", ", said the ", animal),
      12, 0);
```

These two statements will display the exact same line of text on the screen. If animal is the string "cat", the text that they display will be:

```
"Hello", said the cat
```

Cond

Description: Returns the result of one of two expressions depending upon the result of a conditional expression.

Returns: Varies

Usage: Script or steady state.

Function Groups: Logic Control

Related to: Case | IfElse | IfThen

Format: Cond(Condition, Value1, Value2)

Parameters:

Condition

Required. Any numeric expression giving the conditional value to use for the test. If Condition is true

(i.e. not equal to zero), the value of the Value1 parameter is returned.
If Condition is false (i.e. equal to zero), the value of the Value2 parameter is returned.
If Condition is invalid, the return value is invalid.

Value1

Required. Any expression whose value is returned if the Condition parameter is true.

Value2

Required. Any expression whose value is returned if the Condition parameter is false.

Comments:

This function will return either Value1 or Value2 based on the truth of Condition. This means that the function may return a valid result even if one of the Value parameters is invalid (i.e. the one not selected). If Cond is executed in a script, and the Condition parameter selects one of the parameters, the other is not evaluated (unless Condition is invalid, and then neither parameter is evaluated). If Cond is executed in a state, both Value1 and Value2 are evaluated, but only one result is returned. Cond may be used on the left side of an assignment; in that case, Value1 and Value2 may be variables, and one of Value1 and Value2 will receive the assignment (depending on the Condition parameter).
This function can be used extensively to change screen images based upon variable values. For example, it can be used to change a color of a bar if it is used within the Bar statement in the Foreground color parameter.
The return value is optional.

Example:

```
y = Cond(x == 3, 4, 5);
```

Y will be set invalid if x is invalid. If x is 3, y will be 4. Otherwise y will be 5.

```
Cond(x == 3, a, b) = 6;
```

If x is invalid, nothing happens. If x is 3, a will be set to 6, otherwise b will be set to 6.

Configure

(VoiceTalk Module)

Description Is used to define how a speech stream will sound and where it will be heard.

Returns Numeric

Usage Script Only.

Function Groups Speech and Sound

Related to: GetDevices | GetVoices | Reset | ShowLexicon | Speak | SpeakToFile | VoiceTalk

Format VoiceTalkStream\Configure([Voice, Device, Rate, Volume])

Parameters

VoiceTalkStream

Required. A speech stream returned from VoiceTalk.

Voice

Optional Specifies the name of the voice in which to speak. This should be one of the voice names returned from a VoiceTalk\GetVoices call, or Invalid to not change the voice.

Device

Optional Specifies where to play back the voice. This can either be a system device ID (), or the name of a device (such as returned from a call to VoiceTalk\GetDevices()).
-1 is the default – usually speaker audio

Rate

Optional Specifies the speed at which the voice is to speak. Normal speed is "0". Valid values are from -10

(very slow) to 10 (very fast). The default is "0".

Volume

Optional Specifies the volume at which the voice is to speak. Valid values are from 0 (silent), to 100 (full speaker volume). The default is 100.

Comments A speech stream can be configured to sound differently by changing the rate and volume parameters. To configure output to go to another device, such as a modem, change the Device parameter.
If a voice or device is specified which is not valid for this stream, or a problem is encountered while trying to change the stream configuration, the stream will be left unchanged.
"VTSFileOutput" is a valid device name. When used, the Speak() module will no longer work, but the SpeakToFile() module can be used to speak a phrase into a .wav file.
This return value is the Windows error code for the problem encountered, or zero if there was no problem.

Note: Will compile and work once if called from Steady State – this is not recommended practice.

ConnectToMachine

(RPC Manager Library)

Description: This subroutine increments the usage count on the specified workstation and forces RPC Manager to attempt to establish a connection with the specified workstation if it is not already connected. Subroutine call only.

Returns: Socket node (warning – see note in comments)

Usage: Script Only.

Function Groups: Network

Related to: DisconnectFromMachine | GetServer | GetServersListed | GetStatus | IsClient | IsPotentialServer | IsPrimaryServer

| Register (RPC Manager) | Send | SetRemoteValue

Format: \RPCManager\ConnectToMachine(Workstation)

Parameters:

Workstation

Required. Any text expression giving the name or IP address of the workstation to which the connection is to be made.

Comments: This subroutine is a member of the RPC Manager's Library, and must therefore be prefaced by \RPCManager\, as shown in the "Format" section. If the application you are developing is a script application, the subroutine call must be prefaced by System\RPCManager\, and the System variable must be declared in AppRoot.src.
While the usage count of the workstation is non-zero, RPC Manager will continue to attempt to establish a connection to the remote workstation.
It is critical that each ConnectToMachine call should be paired with a DisconnectFromMachine call; if the number of ConnectToMachine calls exceeds the number of DisconnectFromMachine calls, the RPC Manager will not behave as expected and disconnection from the remote workstation may be impeded. An unexpectedly positive value for the Srv value in the socket's entry in the RPC Diagnostics Window may be an indication of a ConnectToMachine/DisconnectFromMachine mismatch.
This subroutine doesn't return a (reliable) socket node (i.e. may go invalid). Also, it isn't a socket, it is a socket node instance.

Example:

```
If ! tryToOpen;
[
   tryToOpen = 1;
   sNode = \RPCManager\ConnectToMachine("TestMachine");
]
If sNode\socketOpen Continue;
```

This code snippet opens a socket to the workstation called TestMachine and waits until the connection has been established before continuing.

Related Functions:

You may also refer to "RPC Manager Service" for a listing of Service Control Methods, RPC Methods, and Deprecated RPC Methods.

ConstCount

Description: Returns the number of constant parameters in a function.

Warning: This function should be used by advanced user only.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Compilation and On-Line Modification, Advanced Module

Related to: Compile | FWrite | Save

Format: ConstCount(OpCode)

Parameters:

OpCode

Required Any numeric expression for the opcode of the function.

Comments: This function is used by the compiler to compile functions such as FWrite and Save.

ConvertTimeStamp

Description: Converts a time stamp from one time zone to another.

Returns: Numeric (double)

Usage: Script only.
May be used in optimized Tag Parameter Expressions.

Function Groups: Time and Date

Related to: TimeZone | TimeZoneList

Format: ConvertTimeStamp(Timestamp, SourceTZ, InDST, DestTZ)

Parameters:

Timestamp

Required The timestamp (in seconds) to be converted since midnight of January 1, 1970. The timestamp parameter is limited to a minimum of 0, and a maximum of 34359738367. Values outside this range will cause Invalid to be returned.

SourceTZ

Required. The name of the time zone from which Timestamp originated. The name must be a name in the list returned by the TimeZoneList function. May also be a structure – see Comments.
If Invalid, or not a valid time zone name, then UTC is used as the source time zone. If set to "0", the local time zone is used.

InDST

Required. A flag that indicates whether daylight savings time (DST) was in use in the source time zone at the time indicated by Timestamp.
This flag is only used for the period at the end of DST where a local time may appear twice.
If true (non-0), DST is in effect. If false (0), DST is not in effect.
If invalid, the default value is false.
If the source time zone does not observe DST (as is the case with values stored using UTC, which is everything that the Historian saves), then this flag has no effect.

DestTZ

Required The name of the time zone to which Timestamp is to be converted. The name must be a

name in the list returned by the TimeZoneList function. May also be a structure – see comments.
If invalid, or not a valid time zone name, then UTC is used as the destination time zone. If set to "0", the local time zone is used.

Comments: The function returns the converted timestamp as a number indicating the number of seconds since midnight of January 1, 1970. The SourceTZ and DestTZ parameters must use names as returned by the TimeZoneList function.

The function uses the time zone information in the registry to determine local time zone bias, DST bias, and starting and ending dates of DST. If the information in the registry is incorrect, then it may be updated using the TZEdit tool available from Microsoft.

Both the SourceTZ and DestTZ parameters may be provided as structures. These structures will have two members

- StdTimeZone – a string that represents the standard name for the time zone
- ObservesDST – a Boolean, where 0 indicates that the time zone does not observe daylight savings time and a 1 indicates that it does

The purpose of this structure is to handle systems that are set to a time zone that does normally observe DST, but the users have configured Windows™ to not adjust for daylight savings time. This structure is the same as that used by option 3 of the TimeZone function.

Example:

```
Init [
  If 1 Main;
```

```
  [
    { Convert from UK to Atlantic Standard Time }
    Timestamp = ConvertTimeStamp(CurrentTime(), "GMT Standard Time",
                                 0,  "Atlantic Standard Time");
  ]
]
```

To convert the timestamp stored in SomeStartTime to GMT time:

```
TimeStamp = ConvertTimeStamp(SomeStartTime, TimeZone(3), 0, Invalid);
```

ConvertToDbDate

(ODBC Manager Library)

Description Provides a conversion of a date value into the format used by the indicated database.

Returns The date value, converted to the specified database format

Usage Script only.
May be used in optimized Tag Parameter Expressions.

Related to: ConvertToDbTime | ConvertToDbTimeStamp

Format \ODBCManager\ConvertToDbDate(dbType, ValueIn[, NoDelimiters])

Parameters

dbType

Required numeric value, indicating the type of this DB connection.

DBType	Meaning
0	MS SQL
1	MS Access
2	Oracle
3	MySQL
4	SyBase

ValueIn

Required. Numeric time value expressed in days since January 1, 1970

NoDelimiters

Optional. Any Boolean expression. If this parameter evaluates to true (non-zero), then text delimiters will be suppressed in the output value.
Defaults to true

Comments This module is a member of the ODBCManager Library, and must therefore be prefaced by \ODBCManager\, as shown in "Format" above.

ConvertToDbTime

(ODBC Manager Library)

Description: Provides a conversion of a time value into the format used by the indicated database.

Returns: The time value, converted to the specified database format

Usage: Script only.
May be used in optimized Tag Parameter Expressions.

Related to: ConvertToDbDate | ConvertToDbTimeStamp

Format: \ODBCManager\ConvertToDbTime(dbType, ValueIn[, NoDelimiters])

Parameters:

dbType

Required numeric value, indicating the type of this DB connection.

DBType	Meaning
0	MS SQL
1	MS Access
2	Oracle
3	MySQL
4	SyBase

ValueIn

Required. Numeric time value expressed in seconds since midnight

NoDelimiters

Optional. If this parameter evaluates to true (non-zero), then text delimiters will be suppressed in the output value. Defaults to true

Comments: This module is a member of the ODBCManager Library, and must therefore be prefaced by \ODBCManager\, as shown in "Format" above.

ConvertToDbTimeStamp

(ODBC Manager Library)

Description: Provides a conversion of a timestamp value into the format used by the indicated database.

Returns: The time value, converted to the specified database format

Usage: Script only.
May be used in optimized Tag Parameter Expressions.

Related to: ConvertToDbTime | ConvertToDbDate |

Format: \ODBCManager\ConvertToDbTime(dbType, ValueIn[, NoDelimiters])

Parameters:

dbType

Required numeric value, indicating the type of this DB connection.

DBType	Meaning
0	MS SQL
1	MS Access
2	Oracle
3	MySQL
4	SyBase

ValueIn

Required. Numeric value expressed as a VTScada timestamp

NoDelimiters

Optional. If this parameter evaluates to true (non-zero), then text delimiters will be suppressed in the output value. Defaults to true

Comments: This module is a member of the ODBCManager Library, and must therefore be prefaced by \ODBCManager\, as shown in "Format" above.

ConvertToVTSDate

(ODBC Manager Library)

Description: Provides a format conversion of a date value from that used by the indicated database to the format used by VTScada.

Returns: The date value, converted to the standard VTScada format

Usage: Script only.
May be used in optimized Tag Parameter Expressions.

Related to: ConvertToVTSTime

Format: \ODBCManager\ConvertToVTSDate(dbType, ValueIn)

Parameters:

dbType

Required numeric value, indicating the type of this DB

DBType	Meaning
0	MS SQL
1	MS Access
2	Oracle
3	MySQL
4	SyBase

DBType	Meaning
0	MS SQL
1	MS Access
2	Oracle
3	MySQL
4	SyBase

ValueIn

Required. Date value as used by the indicated database.

Comments: This module is a member of the ODBCManager Library, and must therefore be prefaced by \ODBCManager\, as shown in "Format" above.

ConvertToVTSTime

(ODBC Manager Library)

Description: Provides a format conversion of a time value, from that used by the indicated database, to that used by VTScada

Returns: The time value, converted to the standard VTScada format

Usage: Script only.
May be used in optimized Tag Parameter Expressions.

Related to:

Format: \ODBCManager\ConvertToVTSTime(dbType, ValueIn)

Parameters:

dbType

Required numeric value, indicating the type of this DB connection.

DBType	Meaning
0	MS SQL
1	MS Access
2	Oracle
3	MySQL
4	SyBase

ValueIn

Required. Time value as used by the indicated database.

Comments: This module is a member of the ODBCManager Library, and must therefore be prefaced by \ODBCManager\, as shown in "Format" above.

ConvertToVTSTimeStamp

(ODBC Manager Library)

Description: Provides a format conversion of a time value from that used by the indicated database into a VTScada time stamp format

Returns: The time and date value, converted to a VTScada timestamp

Usage: Script only.
May be used in optimized Tag Parameter Expressions.

Related to: BuffRead | ConvertToVTSTime | ConvertToVTSDate

Format: \ODBCManager\ConvertToVTSTime(dbType, ValueIn[, FormatStr])

Parameters:

dbType

Required numeric value, indicating the type of this DB connection.

DBType	Meaning
0	MS SQL
1	MS Access
2	Oracle
3	MySQL
4	SyBase

ValueIn

Required. Time and date value as used by the indicated database.

FormatStr

Optional. Provides a format to use. Please refer to the BuffRead function for a description of the formatting options

Comments: This module is a member of the ODBCManager Library, and must therefore be prefaced by \ODBCManager\, as shown in "Format" above.

Coordinates

Note: Deprecated. Do not use in new code.

Description: Sets the VTScada screen coordinate limits (also called "world coordinates") used by the graphics functions.

Returns: Nothing

Usage: Script Only.

Function Groups: Graphics, Window

Related to: CoordToPixel | Window

Format: Coordinates(Left, Bottom, Right, Top)

Parameters:

Left

Required. Any numeric expression giving the x screen coordinate of the left side of the window.

Bottom

Required. Any numeric expression giving the y screen coordinate of the Bottom side of the window.

Right

Required. Any numeric expression giving the x screen coordinate of the Right side of the window.

Top

Required. Any numeric expression giving the y screen coordinate of the Top of the window.

Comments: Left cannot equal Right, and Top cannot equal Bottom. Left, Right, Top, and Bottom can be any floating point values including fractions. When this statement changes coordinate limits, all active graphics statements are automatically redrawn. It may be useful for setting screen coordinates to match device coordinates, or to ensure the screen coordinates are set correctly when working with different video modes.
If you wish to set to pixel coordinates you may do so using this statement or it may be simpler to set the window option to pixel.

Example:

```
Coordinates(0, 349, 639, 0);
```

This sets the screen coordinates such that (320,175) is the center of the screen. If VTScada is running in EGA resolution (640 x 350) then one screen coordinate is the same as one pixel.

CoordToPixel

Description: Takes a specified coordinate pair within a given window

and returns the overall, onscreen pixel location.

Returns: Numeric

Usage: Script Only.

Function Groups: Graphics, Window

Related to: Coordinates

Format: CoordToPixel(Object, CoordX, CoordY, Option)

Parameters:

Object

Required. The window context that the coordinates are taken from.

CoordX

Required. Any numeric expression giving the x-coordinate to convert.

CoordY

Required. Any numeric expression giving the y-coordinate to convert.

Option

Required. Any numeric expression specifying the converted coordinate to return, where 0 returns the converted x-coordinate, and 1 returns the converted y-coordinate.

Comments: This function's return value is determined by the Option parameter.

Example:

```
If 1 Main;
[
  xPixel = CoordToPixel(Self(), xCoord, yCoord, 0);
  yPixel = CoordToPixel(Self(), xCoord, yCoord, 1);
]
```

This converts the point specified by the user coordinates (xCoord, yCoord) in the current module to screen coordinates (pixel location).

CopyDir

(System Library)

Description: This subroutine recursively copies a directory's files and sub-directories down through the entire directory tree.

Returns: Nothing

Usage: Script Only.

Function Groups: File I/O

Related to: CheckFileExist | CheckPathExist | GridList

Format: \System\CopyDir(Destination, Source)

Parameters:

Destination

Required. Any text expression giving the directory into which the source directory is to be copied. This dir-ectory must already exist.

Source

Required. Any text expression giving the directory to be copied.

Comments: This subroutine is a member of the System Library, and must therefore be prefaced by \System\, as shown in the "Format" section. If you are developing a script application, use "System\..." rather than "\System\..." in the function call.

Example:

```
System\CopyDir("C:\OEMApp", "App1");
```

CopyIn

Description: Copies data from an absolute RAM address and returns a buffer.

Returns: Buffer

Usage: Script Only.

Function Groups: Memory I/O

Related to: CopyOut | MemIn | MemOut

Format: CopyIn(Address, Length)

Parameters:

Address

Required. Any numeric expression giving the absolute RAM address to start to copy. Typically the address is specified using the colon (:) operator.

Length

Required. Any numeric expression giving the number of bytes to copy.

Comments: This function requires that the VTSIO driver be installed. Please refer to the topic, Communicating Directly With Hardware for more details.
A buffer is returned that is a copy of RAM at Address.

Example:

```
buff = CopyIn(0x1859@0x0000, 256);
```

This creates a 256 byte text buffer which is a copy of the real-mode RAM bytes at segment 1859 (hex), offset 0 to offset FF (hex).

CopyObjects

Description: Copies the code for selected drawing objects and returns it in a buffer.

Returns: Buffer

Usage: Script Only.

Function Groups: Graphics

Related to: PasteObjects

Format: CopyObjects(EditFAB)

Parameters:

EditFAB

The window where the dragging/drawing of the object is done.

CopyOut

Description: Copies data from a buffer to an absolute RAM address.

Returns: Nothing

Usage: Script Only.

Function Groups: Memory I/O

Related to: CopyIn | MemIn | MemOut

Format: CopyOut(Buffer, Address, Length)

Parameters:

Buffer

Required. Any text or buffer expression containing data bytes to copy.

Address

Required. Any numeric expression giving the absolute RAM address to start to copy.

Length

Required. Any numeric expression giving the number of bytes to copy.

Comments: This function requires that the VTSIO driver be installed. Please refer to the topic, Communicating Directly With Hardware for more details.
A buffer is written to RAM at Address. If fewer than Length bytes are in the buffer, the copy will stop when the end of the buffer is reached.

Example:

```
CopyOut(x, 0x1859:0x0000, 256);
```

This would copy the first 256 bytes from the text variable x to real-mode RAM segment 1859 (hex) offset 0 to offset FF (hex), unless x is shorter than 256 bytes.

CopyRecords

(ODBC Manager Library)

Description: Copy record(s) from a database and inserts the values back into the same table with the desired change to one field. It will destroy any existing records with the same name. Should be run as a called module, waiting for completion. Do not call as a subroutine.

Usage: Script Only.

Returns: An error code if one results from the action, otherwise 0.

Related to:

Format: \ODBCManager\CopyRecords(DSN, TableName, KeyField, OrgKeyValue, CopiedKeyValue, Username, Password [, TransactionObj])

Parameters:

DSN

Required. Data source name of the database to use

TableName

Required. Table name to read and write in both databases

KeyField

Required. Field name to use for the SQL WHERE clause

OrgKeyValue

Required. Key values of the records when complete

CopiedKeyValue

Required. Key values of the records to copy

Username

Required. User name for database access

Pass

Required. Password for database access

TransObj

Required. Value of transaction object

Comments: This module is a member of the ODBCManager Library, and must therefore be prefaced by \ODBCManager\, as shown in "Format" above.

Cos

Description: Returns the trigonometric cosine of an angle in radians.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Trigonometric Math

Related to: ACos | ASin | ATan | Sin | Tan

Format: Cos(Angle)

Parameters:

Angle

Required. Any numeric expression giving the angle in radians.

Comments: The returned value is a number in the range of -1.00 to +1.00. To convert an angle from degrees to radians multiply by \pi / 180 or (approximately) 0.0174533.

Example:

```
x = Cos(180 * \pi / 180);
```

The value of x will be - 1.

CoverageSnapshot

Description: Captures the areas in a VTScada source file which have not executed along with summary statistic for each module in the file to extract code coverage information.

Returns: Struct

Usage: Script Only.

Function Groups: Advanced Module

Related to:

Format: CoverageSnapshot(Module, [Reset])

Parameters:

Module

Required. A VTScada module. The data returned will be for the source file which contains that module. Information for all of the modules in the same source file will be included in the returned value. The module need not be the root module of the source file.

Reset

Optional Tells the function to reset all of the code coverage counters to 0 immediately after the statistics are collected.
A true value indicates that the reset should be done and a false (0) indicates that it should not be done. The default value is 1.

Comments: The VTScada interpreter engine keeps counters for every module, variable, state, statement and function parameter. All of these counters start at 0 whenever VTScada starts. Every time one of these internal structures is executed, its corresponding counter is incremented.
The CoverageSnapshot function searches through

the code for a given VTScada script source file and locates all of the counters which are still 0. These represent the areas of the code that have not executed since VTScada started or at least since the last reset.

CoverageSnapshot returns a list of all these areas of the code that haven't executed. The list is in the form of starting and ending byte offsets in the source file. The list excludes constants the declaration of TEST modules and FIXTURES.

The structure returned is

```
FileCoverage Struct [
  Children  { Dictionary of child module names.
The values are
              either a pointer to Child Struct or
a file name.
              File name is used if not in the
same source file.
             File names are relative to VTScada
installation directory.};
  List      { Pointer to an UncoveredList Struct
};
  FileSize { Number of bytes in the file. Will be
invalid if the
             first parameter of the  function
refers to a module
            which is not the root module in the
file and the
             parent of the module is invalid. };
  TimeStamp { File time for the file       };
  Class     { Variable class defined for the root
module
              In the file. };
];
```

The referenced structures are:

```
Child Struct [
  Begin    { Byte offset from the beginning of
the source file for
              module definition };
  End      { Last byte offset for the module
definition    };
  Children { Dictionary as defined in FileCover-
```

```
age Struct    };
  Class    { Variable class defined for the
module           };
];

UncoveredList Struct [
  Begin     { Offset to start of uncovered code
in source file  };
  End       { Offset to last byte of uncovered
code in range    };
  Mode      { 0 for code, 1 for variables
};
  Next      { Pointer to next UncoveredList for
the file        };
]
```

The UncoverList is ordered in increasing Begin offsets with none of the Begin/End areas overlapping in the list.

CRC

Description: Returns the cyclic redundancy check (CRC) value for a buffer.

Returns: Numeric

Usage: Script or steady state.

Function Groups: String and Buffer

Related to: CRCTable

Format: CRC(Buffer, Offset, Length, Table, Start)

Parameters:

Buffer

Required. Any text expression for which to generate a CRC.

Offset

Required. Any numeric expression giving the offset from 0 in the buffer where the CRC calculation will start.

Length

Required. Any numeric expression giving the number of buffer bytes to include in the CRC. Length must not be greater than 65500.

Table

Required. A text expression giving a CRC look-up table buffer. The length of the table buffer must be a multiple of 256 plus 1.
A table can be generated automatically with the CRCTable function.

Start

Required. Any numeric expression giving the initial value for the register used in the CRC calculation. This is 0 for Allen-Bradley® protocols, and FFFF Hex for Modicon®.

Comments: CRC is a driver toolkit function.

Example:

```
checkSum = CRC(response { Text buffer },
               0 { Start of the buffer },
               20 { Number of bytes },
               CheckTable { Look-up table buffer },
               0 { Use Allen-Bradley™ protocols });
```

This gets the checksum for bytes 0 to 19 of the text buffer response, using the CheckTable CRC look-up table, and an initial accumulator value of 0.

CRCTable

Description Returns a buffer containing a CRC table.

Returns Buffer

Usage Script or steady state.

Function Groups String and Buffer

Related to: CRC | CRCTable

Format CRCTable(Polynomial, Length, Shift)

Parameters

Polynomial

Required. A long expression that gives the bit pattern of the generating polynomial. For Allen-Bradley® and Modicon®, Polynomial = 0b1000000000000101, which corresponds to the expression:
x^16 + x^15 + x^2 + 1
To determine the value of Polynomial for a given generator expression, drop the highest order term, and represent each term present by a 1 in the bit position equal to its exponent.
For example, x^16 + x^12 + x^5 + 1 would have a Polynomial value of 0b0001000000100001. The first 1 corresponds to x^12, the second to x^5, and the last to 1 (which is x^0). This polynomial is used by the XMODEM protocol.

Length

Required. Any numeric expression giving the byte length of the CRC accumulator register (usually 2 or 4). The CRCTable buffer returned is 256 * Length + 1 bytes long.

Shift

Any numeric expression that is 0 for right-shifted CRC calculations (such as Allen-Bradley® or Modicon®), and 1 for left-shifted CRC calculations (such as XMODEM).

Comments This function is part of the driver toolkit.

Example:

```
checkTable = CRCTable(0b1000000000000101
                      { Polynomial: x^16 + x^15 + x^2 + 1 },
                      2 { 16 bit CRC register },
                      0 { Right-shifted CRC calculations });
```

This generates a CRC look-up table for use with the CRC function.

CreateModule

Description: Creates a new module and returns a pointer to it.

Returns: Pointer

Usage: Script Only.

Function Groups: Compilation and On-Line Modifications, Advanced Module

Related to: AddModule | DeleteModule

Format: CreateModule(FileName)

Parameters:

FileName

Required. Any text expression giving the name of the .SRC file that contains this module's definition.

Comments: This function creates a module that has no parent.

Example:

```
If 1 Main;
[
 myMod = CreateModule("pump.SRC");
]
```

CriticalSection

Description: Marks a section of a module as a critical section and will not allow interruption of its execution by other threads.

Returns: Nothing

Usage: Script Only.

Function Groups: Compilation and On-Line Modifications, Basic Module

Related to: Execute

Format: CriticalSection(Statement1, Statement2, [Statement3, ...])

Parameters:

Statement1, Statement2, Statement3, ...

Required. Any expressions to be executed. Any number of parameters may be used.

Comments: This statement works in a manner similar to Execute, but on the threaded level.

Example:

```
If 1 Main;
[
  IfThen(Valid(level),
       CriticalSection(temp = a,
                       a = b,
                       b = temp ) );
]
```

Crop

Description: Modifies an existing image, producing a new one that displays a sub-section of the original.

Returns: Image handle

Usage: Script or steady state.

Function Groups: Graphics, Window

Related to: BitmapInfo | GUIBitmap | GUIButton | ImageArray | ImageSweep | MakeBitmap | ModifyBitmap

Format: Crop(OrigBMP, Left, Top, Width, Height)

Parameters:

OrigBMP

Required. The original image to be cropped. This must be a valid image object, created with the MakeBitmap statement.

Left

Required. Any expression for the number of pixels from the left side of the original image to set to be the

left side of the new image. "0" will start at the same place as the original image.

Top

Required. Any expression for the number of pixels from the top of the original image to set to be the top of the new image. "0" will start at the top of the original image.

Width

Required. Any expression for the number of pixels wide the new image will be. The new image will be enforced to be at least 1 pixel wide, even if Width is set to less than 1 pixel.

Height

Required. Any expression for the number of pixels high the new image will be. The new image will always be at least 1 pixel height, even if Height is set to less than 1 pixel.

Comments:

This function allows for the enlargement of parts of an original image, without requiring that the entire image be drawn. It also uses very little memory, and is very fast, as compared to calls to creating a file from a disk file each time.

There are no restrictions on how large the new image may be, so it can be larger (in any dimension) than the original. If this is the case, the extra space will be filled with black pixels. Values for Left and Top that are less than zero (0) are therefore permitted. Width and Height should be given a value of at least one "1", but may be as large as desired. Any transparency in the original image is preserved in the cropped image.

Cropping can be cumulative, so that you can take the output of one crop function and use that as input to another. The result is that the left and top of the new transform are

equivalent to the sum of the values in each of the original transforms. However, since width and height are in pixels, the number of pixels specified each time is exactly how large the resulting image will be.
The image resulting from a call to Crop() can be used any place that an image may be used, such as in a call to GUIBitmap or GUIButton.

Example:

```
img = Crop(MakeBitmap("TANKS.BMP", 0), 10, 10, 50, 50);
           GUIBitmap(250, 150, 350, 50 { position },
           1, 1, 1, 1, 1 { scaling },
           0, 0 { trajectory, rotation },
           1, 0   { visibility },
           0, 1, 1  { selectability },
          img { image });
```

This example displays a 50x50 pixel subsection beginning 10 pixels from the left and 10 pixels from the top of the original image, "TANKS.BMP". It is then displayed in a 100x100 pixel square on the screen. Any place the color was black (index 0) in the image will be shown as transparent.

```
If Watch(1, BitmapFile);
[
 img = MakeBitmap(BitmapFile);
  imgWidth = BitmapInfo(img, 0);
  imgHeight = BitmapInfo(img, 1);
]
img2 = Crop(img, imgWidth*0.25, imgHeight*0.25,
            imgWidth*0.5, imgHeight*0.5);
```

This example creates and stores an image in img2 which is the middle section of the original image.

CrossReference

Description:	Returns a linked list of structures representing all references to a specified variable or module.
Returns:	Linked list of structures
Usage:	Script Only.

Function Groups: Advanced Module

Related to:

Format: CrossReference(ModuleToSearch [, WhatToFind, Context])

Parameters:

ModuleToSearch

Required. Module value which is the root of the static module tree to search. May also be an array of module values.
If the next parameter (ModuleToFind) is invalid, then only the variable references which are not within the scope of this module are returned, i.e. external references.

WhatToFind

Optional. May be either a module or a variable to search for. If invalid, all references to all variables (whether they may contain modules or otherwise) outside the specified scope tree are returned.

Context

Optional. Instance of a module that indicates what scope ModuleToSearch would run in, were it to run. (Note that ModuleToSearch does not need to be running when CrossReference is called.)
If invalid, only the static scope will be searched.

Comments: Context must be valid where scoping needs to be resolved as shown in the example. In this example, it is necessary to determine if the Draw module referred to is the correct one based up on the module value of the actual Draw variable stored in Tag23's module.

Example:

Assume that there is page named MyPage whose only line is the following:

```
Variable("Tag23")\Draw(Variable("Flow")\Value * 10, 57)
```

Further, assume that Tag23 is of type MyTagType
Then, to look for all references to MyTagType's Draw method inside a particular page, use:

```
CrossRefData = CrossReference(\MyPage, \ MyTagType \Draw, \Code);
```

You could then expect the following to be true.

```
CrossRefData\Reference == FindVariable("Draw", \MyTagType, 0, 0)

CrossRefData\DAG == the code value for the Call to the Draw method
CrossRefData\Begin == the number of bytes from the beginning of the
                      Function to the beginning of the text "Draw".
CrossRefData\End == the number of bytes from the beginning of the
                      Function to the end of the text "Draw".
CrossRefData\Next = Invalid, assuming there are no other references
                    to MyTagType's Draw method on the page.
```

If you wanted to find all the external references in a section of code (i.e. all references to values that aren't actually declared in the MyPage module), then you could use the function like so:

```
CrossRefData = CrossReference(\MyPage, Invalid, \Code);
```

You would expect the linked list in CrossRefData to have nodes for each of the following: Tag23, Draw, Flow, and Value since they are all variables that were not declared in MyPage.
If, for example, the CrossReference code was not able to find a variable called Flow in \Code, then the Reference value for that node in the CrossRefData would be the text "Flow" rather than a variable value for Flow. This is because there simply wasn't a variable named Flow declared in the app. The node for Value would also have the text "Value" instead of a variable value, because if it cannot find the variable for Flow, it follows that it won't find Flow\Value.
The linked list of structures, as returned by this function, are ordered such that the Begin fields are in strictly descending order for each unique file. This is because, if the list is processed in the order returned then multiple search and replace processes done within the same file will not interfere with each other .

All structures for a given file are consecutive in the linked list. In other words, the last reference in the source file will be the first one on the list.
The following tasks can be facilitated without the need to run the code being examined:

- Changing the name of a variable or module
- Moving a variable/module to a different scope
- Removing references to a deleted variable/module
- Giving a cross-reference to locations where a variable/module is referenced
- Re-ordering the parameters to a module
- Locating all variable/module references in a higher scope

CurrentLine

Note: Deprecated. Do not use in new code.

Description: Returns the text string that is the current line in an editor.

Returns: Text

Usage: Script or steady state.

Function Groups: Editor

Related to: AddEditorText | Editor | ForceEvent | GoToOffset | MakeEditor | SetEditMode

Format: CurrentLine(Editor)

Parameters:

Editor

Required. Any expression that returns an editor value that was created by the MakeEditor function. If this isn't an editor type value, or if it is invalid, nothing happens.

Example:

```
If ZButton(10, 40, 110, 10, "Copy", 1, 0);
[
```

```
  lineToCopy = CurrentLine(myEditor);
]
```

CurrentTime

Description: Returns the number of seconds, in local or UTC time, since midnight of January 1, 1970 (where "midnight" is 00:00).

Returns: Numeric (double)

Usage: Script only.
May be used in optimized Tag Parameter Expressions.

Function Groups: Time and Date

Related to: Now | Seconds | Today | TimeArrived

Format: CurrentTime([TimeType])

Parameters:

TimeType

Any expression that evaluates to a 0 or 1. When 0, it indicates that time returned should be local time.
If set to 1, indicates that the current UTC time should be returned.
Defaults to 0 if missing or Invalid.

Comments: This function solves the problem encountered when using Seconds together with Today to determine the current date and time, when the time is within a fraction of a second of midnight.
This function assumes that "midnight" on January 1, 1970 is 00:00, rather than 24:00 on a 24-hour clock.
The returned value is accurate to three decimal points (milliseconds).

Example:

```
If ! Valid(RightNow);
[
  RightNow = CurrentTime();
]
```

This assigns the current time in seconds since midnight on January 1, 1970 to the double variable rightNow.

CurrentWindow

Description:	Returns the application window over which the mouse cursor rests.
Returns:	Object Value
Usage:	Steady State only.
Function Groups:	Locator, Window
Related to:	ActiveWindow | WinLocSwitch | WinXLoc | WinYLoc | Window
Format:	CurrentWindow( )
Parameters:	None
Comments:	The return value is the object value of the module instance running in the window that the mouse cursor is over; this is not necessarily the active window. If more than one module instance is running in that window, the return value will be the highest scope module instance running in that window; which should either call, or be the parent or ancestor of, all other module instances in that window. Unlike the ActiveWindow function, CurrentWindow will recognize and return the object value of the root module instance in child windows when the mouse passes over them. This function is useful for things like status lines, or help windows; it only works on windows that belong to VTScada.

Example:

```
obj = CurrentWindow();
winX = WinXLoc(obj);
winY = WinYLoc(obj);
winB = WinLocSwitch(obj);
```

This shows how to get information about the mouse in another window. This window is determined to be the window the mouse is passing over. WinX and winY have the mouse location (which works like XLoc and YLoc), and winB has the current mouse button combination (like LocSwitch).

D Functions

The sections that follow identify all VTScada functions beginning with "D".

Date

Description: Returns a text string giving the date that corresponds to the number of days since January 1, 1970.

Returns: Text

Usage: Script or steady state.

Function Groups: Time and Date

Related to: DateNum | Day | Month | Now | Time | Today | Year

Format: Date(Day, DateForm [, Flags])

Parameters:

Day

Required. Any numeric expression giving the number of days since January 1, 1970. This is a "Julian" style date. The function, Today() is commonly used.

DateForm

Required. Any numeric or text expression selecting the format for the date format. If DateForm is numeric, the

format for the date will be interpreted according to the standard predefined Date Codes.
If DateForm is a text value, it is interpreted as a date formatting string. Please refer to the Date Formatting Strings in the appendix. Note that these key stings are case sensitive.

Note: In the event that the DateForm parameter does not resolve to either a numeric or text value, the system-configured date format, as specified through the Windows Control Panel, is used. In this case, the Flags parameter is used to select from a number of options for the date.

Flags

An optional parameter that is only used in the event that the DateForm parameter does not resolve to a numeric or a text value. The Flags parameter may be set as follows to adjust the format of the date.

Flags	Description
1	Generate the configured short form of the date (e.g. "29/03/04").
2	Generate the configured long form of the date (e.g. "29 March 2004").
8	Generate the configured year/month format (e.g. March 2004). Please note that this option is only available on Windows 2000/XP

Comments: This function is primarily used to convert dates from historical data files to a format that is more easily readable. The Julian style date is used since it

gives an easy method of calculating the time between dates and it is compact for historical records.
If Date is used with a predefined date code, then the result will always be in English, regardless of any system settings.
If Date is used with a format string, such as "dd MMMM yyyy", then the result will be in the user's locale. For example, "25 Febrero 2014"

Example:

```
ZText(10, 590, Date(Today(), 4), 15, 0);
```

This displays today's date in the lower left of the screen.

DateNum

Description: Returns the number of days since January 1, 1970 for a given date.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Time and Date

Related to: Date | Day | Month | Today | Year

Format: DateNum(Day, Month, Year)

Parameters:

Day

Required. Any numeric expression giving the day of the month for the date.

Month

Required. Any numeric expression giving the month number for the date. January is month 1.

Year

Required. Any numeric expression giving the year of the date. The full four-digit date must be used.

Comments: This function performs the inverse function to the Day, Month, and Year functions. No checks are done to verify that the parameters are in a valid range.

Example:

```
numDays = DateNum(16, 08, 1997);
```

NumDays is set to 8394, which is the number of days between 1 January 1970 and 16 August 1997.

DateSelector

Description: Displays a calendar, from which operators can select a date.

Returns: Timestamp (within a parameter).

Usage: Steady State only.

Function Groups: Time and Date, Graphics

Related to: Date | DateNum | Today

Format: DateSelector(ShowSelector, Left, Top, ptrNewDate)

Parameters:

ShowSelector

Numeric. Used as a flag to enable and disable the display of the selector. Also used as a return value: 0 indicates that a date was selected, -1 indicates that the operator closed the dialog without selecting a date

Left

Any numeric expression specifying the location of the left edge of the calendar

Top

Any numeric expression specifying the location of the

top edge of the calendar.

ptrNewDate

A pointer to a variable holding a timestamp value. The returned date from the calendar will be found here. You may also use this to hand an initial date to the calendar, to be displayed when ShowSelector becomes true

Comments: The function returns an object variable of itself upon starting, in case the calling module requires it.
You can monitor the value of ShowSelector, after making it true to show the calendar, in order to know whether the operator selected a date (ShowSelector becomes 0) or closed the calendar without choosing a new date (ShowSelector becomes -1).
Note that the value within ptrNewDate is a timestamp in seconds, not a date value in days.

Examples:

```
If !Valid(StartDate);
[
   StartDate = Today() * 86400;
]
If GUIButton(0, 1, 1, 0,
             1 - (MID - 60), TOP + 47,
             Split, 1 - (TOP + 17),
             1, 0, 0, 1, 0,
             68, 2, 0,
             \ButtonFace, \ButtonHighlight, \ButtonShadow, \But-
tonTextColor,
             0, 0,
             "Change", "Change", \_DialogFont,
              2, 1, 0);
[
   ShowCalendar = 1;
]
\DateSelector(ShowCalendar, LHS, TOP + 60, &StartDate);
```

Day

Description: Returns the day of the month for a given date number.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Time and Date

Related to: Date | DateNum | Month | Today | Year

Format: Day(Date)

Parameters:

Date

Required. Any numeric expression giving the number of days since January 1, 1970.

Comments: This function works in conjunction with the Month and Year functions to decompose a date into the corresponding day, month and year.

Example:

```
firstDate = Day(DateNum(25, 12, 1992));
```

This causes firstDate to be set to 25.

DBAdd

Description: Executes in its own thread to add a record to a VTScada database and returns an indication of parameter errors.

Returns: Numeric

Usage: Script Only.

Function Groups: Database and Data Source

Related to: DBAdd | DBGetStream | DBListGet | DBListSize | DBRemove | DBSystem | DBTransaction | DBUpdate | DBValue

Threaded: Yes

Format: DBAdd(DBSysVal, IDKey, DefaultEvent, EventCode [, FieldVal])

Or

DBAdd(DBSysVal, IDKey, DefaultEvent, EventCode [, FieldVal1, FieldVal2, FieldVal3, …])

Parameters:

DBSysVal

Required. The database value to use. This is the return value from a DBSystem call.

IDKey

Required. Any text expression that uniquely identifies the record to be created.

DefaultEvent

Required. Any text expression containing one byte for each list in the system. Any missing bytes or bytes with values not included in the list provided, default to "do nothing to the list". Note that there is one byte for each list in the system.

Text Value	Default Event
+	Add to list
<space>	Do nothing to list

EventCode

Required. Any variable in which the event code for the function is returned. This parameter may be used to indicate completion of the function, as it will not be set to a valid value until execution is complete. It has one of the following meanings:

EventCode	Meaning
0	New record added
1	Record already existed. it was

	updated
2	Error occurred – record not added

EventCode may be replaced by a constant or Invalid if it is not required. This parameter will not be set if there is a parameter error (see comments)

FieldVal

(or FieldVal1, FieldVal2, FieldVal3, …)
One or more parameters that are an array or a list of values for each field. FieldVal (or FieldVal1, FieldVal2, FieldVal3, …) will be set to Invalid if not specified, if insufficient parameters are specified to fill each field in the new database record, or if the parameter is an array with insufficient entries to fill each field in the database.

Comments:

This function executes in the thread created by the DBSystem call, so it will not block other statements from executing. This does mean, however, that the timing for EventCode becoming valid (marking that the statement has finished executing) is unpredictable and should therefore be checked for validity prior to executing other statements that rely on this statement's results.
The return value for this function indicates if any of its key parameters (DBSysVal, IDKey, DefaultEvent, or any required FieldVal) are illegal. It will immediately return a value of false (0) unless a key parameter was illegal, in which case it will return true (1). Note that the return value only signals completion of the function's execution if it is true, otherwise the function will continue executing in the thread created for it.
If the record that is to be added already exists (i.e. its IDKey duplicates one already in the system), the field val-

ues of the pre-existing record are updated by those defined in the FieldVal parameter(s).
If the database file has its read-only attribute set when this function is executed, it will be cleared automatically by execution of the function.

Example:

```
alarmDB = DBSystem("", "", 0, 0, 32 { key }, -3 { pressure },
3 { level }, -3 { temperature });
If Valid(AlarmDB) && ! added;
[
  added = 1;
  DBAdd(AlarmDB { Database to use },
      "Tank_697" { ID key },
       "+" { Add to list },
       event { Var to get event code },
       pressure, level, temperature
  { Field values }));
]
```

DBDropList

(ODBC Manager Library)

Description: Populates a droplist using the results of an SQL query on a given database.

Format: \ODBCManager\DBDropList(X1, Y1, X2, Y2, DSN, UserName, Password, SQLQuery, Title, LocFocusID, Init, Var[, DrawBevel, AlignTitle, AddInvalid, InvalidText])

Returns: Numeric

Usage: Script Only.

Function Groups: Database and Data Source

Related to: Droplist |

Parameters:

X1

Required. Screen coordinates of the left side of drop list

Y1

Required. Screen coordinates of the top of list

X2

Required. Screen coordinates of the right side of the list

Y2

Required. Screen coordinates of the bottom of list

DSN

Required. DSN of the database to query

UserName

Required. The user name in the database for authentication.

Password

Required. The password in the database for authentication.

SQLQuery

Required. An SQL query that returns either one or two columns. The first column is used for droplist display, the second (if it exists) provides the matching value for each entry in the list.

Title

Required. The title for the drop list

LocFocusID

Required. Numeric expression for the focus number of this graphic. If this value is 0, the droplist will display its current setting, but will not be able to be opened (i.e. its value cannot be changed) and will appear grayed out. The default value is 1.

Init

Required. Any expression for the initial value displayed in the field

Var

Required. A variable whose value will be set by this droplist

DrawBevel

Optional. If true (non-0) a bevel is drawn around the droplist. If false (0) no bevel is drawn. The default value is false

AlignTitle

Optional. If true (non-0) the title is included in the calculation for vertical alignment. If false(0) it is added to the droplist after it (and its bevel if one exists) has been vertically aligned. The default is true

AddInvalid

Required. If true an entry with an invalid value will be added at the top of list.

InvalidText

Required. Used with AddInvalid to provide the text to be displayed for the item at the top of the list.

Comments: This module is a member of the ODBCManager Library, and must therefore be prefaced by \ODBCManager\, as shown in "Format" above.

DBGetStream

Description: Converts a database to a stream, and returns an indication of parameter errors.

Returns: Numeric

Usage: Script

Function Groups: Database and Data Source, Stream and Socket

Related to: DBAdd | DBInsert | DBListGet | DBListSize | DBRemove | DBSystem | DBTransaction | DBUpdate | DBValue

Threaded: Yes

Format: DBGetStream(DBSysVal, Stream [, Timestamp])

Parameters:

DBSysVal

Required. The database value to use. This is the return value from a DBSystem call.

Stream

Required. Any variable in which the stream containing the database will be returned.

Timestamp

Optional. An optional parameter that is any numeric expression indicating the earliest timestamp to include in the stream. Only database records with date/time values greater than or equal to this parameter will be returned.

Comments: The return value for this function indicates if its key parameter (DBSysVal) is invalid

If DBSysVal is valid, DBGetStream will immediately return a value of false (0).
If DBSysVal is invalid, DBGetStream will not perform the required operation, and will instead immediately return a value of 1.

Note that the return value only signals completion of the function's execution if it is true.
Since the value of the Stream parameter is created from a database's file, the DBSystem function that created the database had to have a valid first parameter (the file name in which the database contents are stored).
If the database file has its read-only attribute set when this function is executed, it will be cleared automatically by execution of the function.

Example:

```
db = DBSystem("", "", 0, 0, 32 { key }, -3 { pressure },
3 { level }, -3 { temperature });
If Valid(db) && ! done;
[
  done = 1;
  DBGetStream(db { Database to use },
  result { Returned stream value });
]
```

DBGridList

(ODBC Manager Library)

Description: Populates a grid list using the results of a database query

Format: \ODBCManager\DBGridList(DSN, UserName, Password, SelectQualifier, Fields, Tables, WhereFields, WhereOperators, WhereValues, WhereAndFlag, UserQuery, ColName, ColFormat, ColCellWidth, RowHeight, TitleHeight, ExtDisplayRef, AutoResizeWidths, OrderBy, LockFirstColParm, SortParm, SelectedRowParm, SelectedColParm, Horiz, Vert, Refresh, DataPtr)

Returns: Numeric

Usage: Script Only.

Function Groups: Database and Data Source

Related to: GridList

Parameters:

DSN

Required. Datasource name to query.

UserName

Required. The user name in the database for authentication.

Password

Required. The user name in the database for authentication.

SelectQualifier

Required. SQL selection qualifier, such as "top 100" etc.

Fields

Required. An array of field names to query for

Tables

Required. An array of tables to query from

WhereFields

Required. An array of fields named for SQL WHERE clause

WhereOperators

Required. An array of Operators for the SQL WHERE clause

WhereValues

Required. An array of Array of values for the SQL WHERE clause

WhereAndFlag

Required. Set 1 for AND of where fields, reset for OR

UserQuery

Required. Overrides Fields, Tables, & Where when defined

ColName

Required. Text array of titles for displaying data

ColFormat

Required. Text array of format qualifiers for each of the values.

ColCellWidth

Required. Array giving the size to use for the cells columns

RowHeight

Required. Height of the data rows. Set to invalid to use the defaults.

TitleHeight

Required. Height of the column titles. Set to invalid to use the defaults.

ExtDisplayRef

Required. Object value of where ColFormat modules located

AutoResizeWidths

Required. Automatically resize the widths to fit the available space

OrderBy

Required. Field names for ORDER BY clause only used

LockFirstColParm

Required. TRUE to lock column 0 while horizontal scrolling

SortParm

Required. Enables grid list sorting

SelectedRowParm

Required. 0 - based value of row of selected item

SelectedColParm

Required. 0 - based value of column of selected item

Horz

Required. Horizontal scroll position

Vert

Required. Vertical scroll position

Refresh

Required. Set flag to re-query data

DataPtr

Required. Pointer to data read from db in [Col][Row]

format

Comments: This module is a member of the ODBCManager Library, and must therefore be prefaced by \ODBCManager\, as shown in "Format" above.

DBInsert

Description: Identical to DBAdd, except that it will not modify an existing record. Executes in its own thread to add a record to a VTScada database and returns an indication of parameter errors.

Returns: Numeric

Usage: Script Only.

Function Groups: Database and Data Source

Related to: DBAdd | DBGetStream | DBListGet | DBListSize | DBRemove | DBSystem | DBTransaction | DBUpdate | DBValue

Threaded: Yes

Format:
DBInsert(DBSysVal, IDKey, DefaultEvent, EventCode [, FieldVal])
Or
DBInsert(DBSysVal, IDKey, DefaultEvent, EventCode [, FieldVal1, FieldVal2, FieldVal3, ...])

Parameters:

DBSysVal

Required. The database value to use. This is the return value from a DBSystem call.

IDKey

Required. Any text expression that uniquely identifies the record to be created.

DefaultEvent

Required. Any text expression containing one byte for each list in the system. Any missing bytes or bytes with values not included in the list provided, default to "do nothing to the list". Note that there is one byte for each list in the system.

Text Value	Default Event
+	Add to list
<space>	Do nothing to list

TimeStamp

Required. The time to be used for all lists that the new record is a member of. Current time will be used if this field is invalid.

EventCode

Required. Any variable in which the event code for the function is returned. This parameter may be used to indicate completion of the function, as it will not be set to a valid value until execution is complete. It has one of the following meanings:

EventCode	Meaning
0	New record added
2	Error occurred – record not added

EventCode may be replaced by a constant or Invalid if it is not required. This parameter will not be set if there is a parameter error (see comments)

FieldVal

(or FieldVal1, FieldVal2, FieldVal3, …)

One or more parameters that are an array or a list of

values for each field. FieldVal (or FieldVal1, FieldVal2, FieldVal3, …) will be set to Invalid if not specified, if insufficient parameters are specified to fill each field in the new database record, or if the parameter is an array with insufficient entries to fill each field in the database.

Comments: This function executes in the thread created by the DBSystem call, so it will not block other statements from executing. This does mean, however, that the timing for EventCode becoming valid (marking that the statement has finished executing) is unpredictable and should therefore be checked for validity prior to executing other statements that rely on this statement's results.

The return value for this function indicates if any of its key parameters (DBSysVal, IDKey, DefaultEvent, or any required FieldVal) are illegal. It will immediately return a value of false (0) unless a key parameter was illegal, in which case it will return true (1). Note that the return value only signals completion of the function's execution if it is true, otherwise the function will continue executing in the thread created for it.

If the database file has its read-only attribute set when this function is executed, it will be cleared automatically by execution of the function.

Example:

```
alarmDB = DBSystem("", "", 0, 0, 32 { key }, -3 { pressure },
3 { level }, -3 { temperature });
If Valid(AlarmDB) && ! added;
[
  added = 1;
  DBInsert(AlarmDB { Database to use },
      "Tank_697" { ID key },
       "+" { Add to list },
       event { Var to get event code },
       pressure, level, temperature
```

```
  { Field values }));
]
```

DBListGet

Description: Executes in its own thread to retrieve certain records from a list in a VTScada database and returns an indication of parameter errors.

Returns: Numeric

Usage: Script Only.

Function Groups: Database and Data Source

Related to: DBAdd | DBInsert | DBGetStream | DBListSize | DBRemove | DBSystem | DBTransaction | DBUpdate | DBValue

Threaded: Yes

Format: DBListGet(DBSysVal, Result, Orientation, List, Start, Number, Method [, Events, IDKey, Filters, Sort, StartTime, StopTime, FieldVals])
(Note that FieldVals may be an array or a series of parameters: Field1, Field2, etc.)

Parameters:

DBSysVal

Required. The database value to use. This is the return value from a DBSystem call.

Result

Required. A variable in which the resulting array will be stored. The dimension of the array will match the number of fields requested.

Orientation

Required. Any logical expression that denotes the orientation of the resultant array. If true (non-0) each record retrieved forms its own row, with each column representing a field. If false (0), the reverse holds true.

This means that with Orientation set to 1, if 5 fields were requested in the FieldVals parameter(s), the resultant array would be R[n][5], where n is the number of matching records found.

List

Required. Any numeric value or array of numeric values that define(s) the list(s) to search.

Value	List
-1	Entire database
0	Transaction log
Number, starting at 1	The specific list number

The database and transaction log cannot be combined with other lists.

Start

Required. Any numeric expression for the first match to include, beginning at 0.

Number

Required. Any numeric expression for the maximum number of matches to return.

Method

Required. Any numeric expression that determines what elements to include in the list. This parameter is one or a combination of the following values

Method	Bit No.	Method Description
0	-	No filtering (include everything)
1	0	Event filtering

2	1	ID key filtering
4	2	Record value filtering
8	3	Sort as per the Sort parameter
16	4	Limits the records returned to the times supplied in the StartTime and StopTime parameters
32	5	Pull records starting from the current offset or record position in the file. Applies only to VTScada database log files and only if they are not sorted.

The filtering/sorting parameters follow (in the specified order), with only those that are applicable being included. If the bit for a certain type of filtering has been set, a parameter corresponding to that option must exist. If a bit is not set, the parameter corresponding to that option should not be included. Setting it to invalid is not acceptable. If a Sort is requested, but the Sort Options parameter is passed in as invalid, then the function will behave as if a sort had not been requested.

Notes for bit 5: do not use a number of matches to skip, even though this would work properly. (That is, starting from the existing position in the log file, n matches would be skipped.) Care must be taken, since if something else changes the current position in the file (a transaction or another DBListGet), then you may not get the records you are expecting.

After the initial DBListGet, subsequent DBListGet calls can set bit 5, adding a value of 32 to the method, to use the offset as it was left by the last pull.

Events

Optional. Used for filtering the records in the list and should only be included if Method designates event filtering (1). This value is a text string containing one byte for each list in the system. Any missing bytes default to all for that list, while bytes with values not included in the following list default to no matches (i.e. nothing will be found).

Note that there is one byte for each list in the system, not for each list that has been selected by the List parameter.

If the transaction log is being used (List = 0), the following codes are compared with the Event field of each record

Events	Added to List	Removed from List	No Changes
0	none (nothing is ever selected)		
1 or +	√		
2 or -		√	
3	√	√	
4 or <Space>			√
5	√		√
6		√	√

7 or *	√	√	√

If one of the lists is being used (List > 0), the following codes are used:

Value	Added to List
0 or 4	None (nothing is ever selected)
1, 5, or +	On the list
2, 6, or -	Not on the list
3, 7, or *	All (everything is selected)

IDKey

Optional. Used for designating the record(s) to use and should only be included if Method designated ID key filtering. It is any text expression and can contain the wildcard characters "*" and "?".

Filters

Optional. Designates the record(s) to use to filter the resulting list by using logical ANDing and ORing and checking for field matches. It should be included only if Method designated record value filtering.

At its simplest, this parameter may hold a 1-dimensional array with 2 or 3 numeric elements.

Element	Description
0	Field to use (-1 filters on date/time values. -2 filters on ID key)
1	Limiting value
2	Comparison value

If (and only if) you are creating an alarm filter that is specifically for use *only* with the alarm name field (\AlarmNameField -2), then two addi-

tional parameters are required, having the following values. This allows the translation of the alarm UniqueID, which is held in the record, to the friendly name used in the search string.

Element	Value
3	\VTSDB
4	\AlarmSeparatorString

The comparison value is a numeric value or expression that indicates the type of comparison to be made, and may be omitted if desired. In this case, the comparison will be taken to mean "is equal to". Valid values for this third element are as follows:

Comparison Value	Comparison	Case Sensitive
0	Equal to	no
1	Greater than	no
2	Less than	no
3	Specified by wildcard (field value is text)	no
4	Not equal to	no
5	Less than or equal to	no
6	Greater than or equal to	no
7	Opposite of wildcard specification (field value is text)	no
8	Equal to	yes
9	Greater than	yes

10	Less than	yes
11	Specified by wildcard (field value is text)	yes
12	Not equal to	yes
13	Less than or equal to	yes
14	Greater than or equal to	yes
15	Opposite of wildcard specification (field value is text) yes	
16	Is the specified bit set? 20 (16 + 4) for not-set.	no

Note that the comparison values of 3, 7, 11, and 15 are only useful when the field value is a text string. For example, if you want to get only those entries whose value starts with "d", the field value should be "d*", and the comparison value should be 3. If you want all entries that don't start with "d", the field value should still be "d*", with a comparison value of 7.

If a more detailed filtering criterion is required, a 2-dimensional array may be used, where additional rows are added, each with the same elements as the first ([n][0] is field number, [n][1] is limiting value, [n][2] is comparison type). All rows will then be ANDed together to form the filtering statement. Once again, the third column may be omitted entirely, however, if it exists it must have valid values in all rows.

The most detailed filtering array occurs when an ORing of field specifications is also required. In this case, this parameter is a 1-dimensional

array, where each element contains a pointer to an array as described previously. The elements in each AND array will be ANDed together, then the results from these ANDed arrays will be ORed.

VTScada differentiates between the various options for this parameter by checking its first element. If it is not a pointer, then the parameter is assumed to contain a single AND array directly (i.e. no OR is performed). It should only be included if Method designated record value filtering.

Sort

Optional. Used for defining the type of sort to be done. Currently, only the bin type of sort is supported where, in a single pass through the array, records are grouped together based on having the same value in the specified field. Sorting may only be performed on numeric data. This parameter is a 1-dimensional array (not one that is created via a New function call), whose elements have the following meanings

Element	Description
0	Method of sorting (1) for bin sort
1	Field number to sort on
2	Flag indicating descending order sort
3	Number of values (bins) to use. Valid range is 1 to 1024
4	Lower limit (start) of range

In the case of elements 3 and 4, if the third ele-

ment is 6, for example, and the fourth element is 5, all records whose specified field has a value from 5 (lower limit) through to 10 (includes 6 elements) will be sorted, while all others will be discarded.

StartTime

Optional. Specifies the oldest record in the file to be used, in the case that bit 4 is set in Method.

StopTime

Optional. Specifyies the newest record in the file to be used, in the case that bit 4 is set in Method.

FieldVals / FieldValN

Optional. A parameter or series of parameters that is either an array or a list of values that indicate which field(s) for which data is requested. If omitted, all fields are returned in their default order. Field numbers range from 1 to 255. The valid values for this parameter are

FieldVals	Field Type Attribute
-2	ID key
-1	Date/time value
0	Event/status for record
1	or more Field value

Note: In the case of an array, it must be a static array – a dynamically declared array (one that is created via a New function call), it will not work here.

Comments: This function executes in the thread created by the DBSystem call, so it will not block other statements from executing. This does mean, however, that the

timing for Result becoming valid is unpredictable and should therefore be checked for validity prior to being used.

The return value for this function indicates if any of its key parameters (DBSysVal, Orientation, List, Start, Number or Method) are invalid.

If all of the key parameters are valid, DBListGet will immediately return a value of false (0).
If any of the key parameters are invalid, DBListGet will not perform the required operation, and will instead immediately return a value of 1.
If the database uses lists then the list parameter may be greater than 0 to select values that appear on that list. For example, when retrieving alarms from their db instance, the following values can be used

Value	List
1	Active alarm list
2	Unacknowledged alarm list
3	Disabled alarm list

For information about the standard alarm lists in VTScada see "Alarm Manager Service".
Note that the return value only signals completion of the function's execution if it is true, otherwise the function will continue executing in the thread created for it.
The dimension of Result will depend on the number of fields requested.
If the database file has its read-only attribute set

when this function is executed, it will be cleared automatically by execution of the function.

Example:

```
dbVal = DBSystem("c:\vts5\app6\equip.db", "", 0, 0, 64 { key },
3 { field 1 }, -2 { field 2 }, -3 { field 3 });
If Valid(dbVal) && ! retrieved;
[
  retrieved = 1;
  DBListGet(dbVal { Database value },
            final { Resultant array },
            0 { Orientation },
            1 { Use entire database },
            0 { Include from first match on },
            20 { Number of matches to get },
            2 { Method - filter by ID key },
            { Events parameter not required }
            "Motor*" { Match with ID key },
            { Filters/Sort parameters not required }
            2 { Get ID key field },
            2 { Get field 2 also });
]
```

If there is any doubt as to the validity of the parameters and further statements rely on final becoming valid, the following version of the script might be more appropriate.

```
If Valid(dbVal) && ! retrieved;
[
  retrieved = 1;
  IfThen(DBListGet(dbVal, final, 0, -1, 0, 20, 2,
         "Motor*", -2, 2),
  final = 0;
  );
]
```

DBListSize

Description: Executes in its own thread to retrieve the size of a certain list in a VTScada database and returns an indication of parameter errors.

Returns: Boolean

Usage: Script Only.

Function Groups: Database and Data Source

Related to: DBAdd | DBInsert | DBGetStream | DBListGet | DBRemove | DBSystem | DBTransaction | DBUpdate | DBValue

Threaded: Yes

Format: DBListSize(DBSysVal, Result, List, Method [, remaining parameters vary according to Method value])

Parameters:

DBSysVal

Required. The database value to use. This is the return value from a DBSystem call.

Result

Required. A variable in which the resulting 1 or 2 dimensional array will be stored.

List

Required. Any numeric value or array of numeric values that define(s) the list(s) to search

List	List Description
-1	Entire database
0	Transaction log
Number, starting at 1	The specific list number

The database and transaction log cannot be combined with other lists.

Method

Required. Any numeric expression that determines how filtering is done. This parameter is one or a combination of the following values

Method	Bit No.	Method Description
0	-	No filtering (include everything)
1	0	Event filtering

2	1	ID key filtering
4	2	Record value filtering
8	3	Place holder for future use
16	4	A supplied date range should be used for filtering.

.

Note that the ability to filter for a supplied date range applies only to VTScada database log files. An example of the format of the supplied date range can be found in the Examples section of this topic.

The filtering parameters follow (in the specified order), with only those that are applicable being included.

Events

Optional. Used for filtering the records in the list and should only be included if Method designated event filtering. This value is a text string containing one byte for each list in the system. Any missing bytes default to all for that list, while bytes with values not included in the following list default to no matches (i.e. nothing will be found). Note that there is one byte for each list in the system, not for each list that has been selected by the List parameter.

If the transaction log is being used (List = 0), the following codes are compared with the Event field of each record

Value	Added to List	Removed from List	No Changes
0	none (nothing is		

	ever selected)		
1 or +	Ö		
2 or -		Ö	
3	Ö	Ö	
4 or <Space>			Ö
5	Ö		Ö
6		Ö	Ö
7 or *	Ö	Ö	Ö

If one of the lists is being used (List > 0), the following codes are used:

Value	Added to List
0 or 4	None (nothing is ever selected)
1, 5, or +	On the list
2, 6, or -	Not on the list
3, 7, or *	All (everything is selected)

IDKey

Optional. Used for designating the record(s) to use and should only be included if the parameter Method designates ID key filtering. This is any text expression and can contain the wildcard characters "*" and "?".

Filters

Optional. Designates the record(s) to use to filter the resulting list by using logical AND or, OR and checking for field matches. It should only be included if the parameter Method designates record value filtering.

At its simplest, this parameter may hold a 1-dimensional array with 2 or 3 numeric elements

Filters	Description
0	Field to use (-1 filters on date/time values)
1	Limiting value
2	Comparison value

If (and only if) you are creating an alarm filter that is specifically for use *only* with the alarm name field (\AlarmNameField -2), then two additional parameters are required, having the following values. This allows the translation of the alarm UniqueID, which is held in the record, to the friendly name used in the search string.

Element	Value
3	\VTSDB
4	\AlarmSeparatorString

The comparison value is a numeric value or expression that indicates the type of comparison to be made, and may be omitted if desired. In this case, the comparison will be taken to mean "is equal to". Valid values for this third element are as follows:

Value	Comparison	Case Sensitive
0	Equal to	no
1	Greater than	no
2	Less than	no
3	Specified by wildcard (field value is text)	no

4	Not equal to	no
5	Less than or equal to	no
6	Greater than or equal to	no
7	Opposite of wildcard specification (field value is text)	no
8	Equal to	yes
9	Greater than	yes
10	Less than	yes
11	Specified by wildcard (field value is text)	yes
12	Not equal to	yes
13	Less than or equal to	yes
14	Greater than or equal to	yes
15	Opposite of wildcard specification (field value is text) yes	
16	Is the specified bit set? 20 (16 + 4) for not-set.	no

Note that the comparison values of 3, 7, 11, and 15 are only useful when the field value is a text string. For example, if you want to get only those entries whose value starts with "d", the field value should be "d*", and the comparison value should be 3. If however, you want all entries that don't start with "d", the field value should still be "d*", with a comparison value of 7.

If a more detailed filtering criterion is required, a 2-dimensional array may be used, where addi-

tional rows are added, each with the same elements as the first ([n][0] is field number, [n][1] is limiting value, [n][2] is comparison type). All rows will then be ANDed together to form the filtering statement. Once again, the third column may be omitted entirely, however, if it exists it must have valid values in all rows.
The most detailed filtering array occurs when an ORing of field specifications is also required. In this case, this parameter is a 1-dimensional array, where each element contains a pointer to an array as described previously. The elements in each AND array will be ANDed together, then the results from these ANDed arrays will be ORed.
VTScada differentiates between the various options for this parameter by checking its first element. If it is not a pointer, then the parameter is assumed to contain a single AND array directly (i.e. no ORing is performed). It should only be included if Method designated record value filtering.

Comments:

This function executes in the thread created by the DBSystem call, so it will not block other statements from executing. This does mean, however, that the timing for Result becoming valid is unpredictable and should therefore be checked for validity prior to being used.
The return value for this function indicates if any of its key parameters (DBSysVal, List, or Method) are invalid.
If all of the key parameters are valid, DBListSize will immediately return a value of false (0).
If any of the key parameters are invalid, DBListSize will not perform the required operation, and will instead imme-

diately return a value of 1.
If no filter being used at all, the function reports the number of records stored in the header of the underlying formatted file class, rather than counting records one by one.
Note that the return value only signals completion of the function's execution if it is true, otherwise the function will continue executing in the thread created for it.
If the database file has its read-only attribute set when this function is executed, it will be cleared automatically by execution of the function.
For information about the standard alarm lists in VTScada see "Alarm Manager Service".

Example:

```
db = DBSystemDBSystem(dbFile, "", 0, 0, 64 { key }, -3 { field 1 });
If Valid(db) && ! gotSize;
[
  gotSize = 1;
  DBListSize(db { Database to use },
  size { Resultant value },
  2 { List number to use },
  0 { Method - include everything }
  { Events parameter not required }
  { IDKey parameter not required }
  { Filters parameter not required });
]
```

If there is any doubt as to the validity of the parameters and further statements rely on size becoming valid, the following version of the script might be more appropriate.

```
If Valid(db) && ! gotSize;
[
  gotSize = 1;
  IfThen(DBListSize(db , size, 2, 0 ),
  size = 0;
  );
]
```

If filtering for a supplied date range:

```
DBListSize(DBLogVal, MatchingRecords, 0 { Log list },  16 { date
range method}, ClientTime, ServerTime)
```

DBRemove

Description: Executes in its own thread to remove a record from a VTScada database and returns an indication of parameter errors.

Returns: Numeric

Usage: Script Only.

Function Groups: Database and Data Source

Related to: DBAdd | DBInsert | DBGetStream | DBListGet | DBListSize | DBSystem | DBTransaction | DBUpdate | DBValue

Threaded: Yes

Format: DBRemove(DBSysVal, IDKey [, EventCode])

Parameters:

DBSysVal

Required. The database value to use. This is the return value from a DBSystem call.

IDKey

Required. Any text expression used for designating the record(s) to delete. Can contain the wildcard characters "*" and "?".

EventCode

Optional. Any variable in which the event code for the function is returned. This parameter may be used to indicate completion of the function, as it will not be set to a valid value until execution is complete. It has one of the following meanings

EventCode	Meaning
0	Record(s) deleted
1	Error occurred – record(s) not

deleted

EventCode will not be set if there is a parameter error (see comments).

Comments: This function executes in the thread created by the DBSystem call, so it will not block other statements from executing. This does mean, however, that the timing for EventCode becoming valid (marking that the statement has finished executing) is unpredictable and should therefore be checked for validity prior to executing other statements that rely on this statement's results.
The return value for this function indicates if any of its key parameters (DBSysVal or IDKey) are invalid. It will immediately return a value of false (0) unless a key parameter was invalid, in which case it will return true (1). Note that the return value only signals completion of the function's execution if it is true, otherwise the function will continue executing in the thread created for it.
If the database file has its read-only attribute set when this function is executed, it will be cleared automatically by execution of the function.

Example:

```
myDB = DBSystem(dbFile, "", 0, 0, 64 { key }, 10 { field 1 });
If Valid(myDB) && ! killed;
[
  killed = 1;
  DBRemove(myDB { Database to use },
           "Digital*" { ID key },
           eCode { Event code returned });
]
```

DBSystem

Description: Creates a VTScada database and returns its value. The maximum field length is 65,523 characters. If the field length is longer than 65,523 characters, the DBSystem call will return invalid.

Returns Numeric

Usage: Steady State only.

Function Groups: Database and Data Source

Related to: DBAdd | DBInsert | DBGetStream | DBListGet | DBListSize | DBRemove | DBTransaction | DBUpdate | DBValue

Format:

DBSystem(DBFile, DBTransFile, MaxTrans, NumLists, IDKeySize, FieldAttribs)
Or
DBSystem(DBFile, DBTransFile, MaxTrans, NumLists, IDKeySize, FieldAttrib1, FieldAttrib2, FieldAttrib3)

Parameters:

DBFile

Required. A text expression giving the file in which to store the database information. If this parameter is invalid or is a null text string (""), no file is created and the database is reconstructed every time the system starts. DBFile can be an array containing 1 or 2 file names or streams. On startup, the second stream will be brought into sync with the first. The first is only read to initialize the database system. Both streams are maintained while the system is running.
Note that the value for this parameter may not be changed while the DBSystem call is active (i.e. the file cannot change) – any changes will be ignored.

DBTransFile

Required. A text expression giving the file in which transactions are logged. If this parameter is invalid or is a null text string (""), no file is created and no transaction logging is done. Unlike the DBFile parameter, the value of this parameter may be changed even while the DBSystem function is active (i.e. the file is dynamic).

MaxTrans

Required. Any numeric expression giving the maximum number of transactions to be stored in the transaction log file.

NumLists

Required. Any numeric expression giving the number of memory lists to be maintained. The valid range is from 0 to 255.

IDKeySize

Required. Any expression that designates the maximum length for the ID key. This affects the size of the files and the database stored in memory.

FieldAttribs

Required. Either an array or a series of parameters that defines what type and size the fields are. The number of parameters or array entries determines how many fields there are. The valid field types are

FieldAttribs	Field Type	Attribute
-4	Indicates a VTScada value field.	Any VTScada value may be written, but will only go into the in-memory copy of the database. The disk record will have a zero length field.
-3	Double precision float	(8 bytes)
-2	Long integer	(4 bytes)
-1	Short integer	(2 bytes)
0	Indicates a Status field	.
1 or more	Text	(number defines length)

Comments:

This function is not threaded, however, it creates a thread inside of which the database value referring to the database system requested is accessed. All other database functions (those beginning with "DB"), except for DBValue, do not create their own thread, but will execute in the thread created by this function. This thread will exist for as long as the DBSystem statement remains active (i.e. until a state

change occurs). For this reason, the state containing the DBSystem call must remain active until all other database statements have finished executing. The statement responsible for a state change should therefore trigger only when all results from all statements accessing this database are valid.

Note: WARNING: There must only be one DBSystem function acting on a file at any given time, whether the file is used as the DBFile or the DBTransFile parameter. If more than one DBSystem function is affecting the same file, it may become corrupted, even if one of the database handles is only used to read information from the database.

The database system produced by this function consists of three primary parts

- A memory database
- A transaction log file
- From 0 – 255 user specified lists

The memory database, as the name implies, is stored only in memory, and is created at the exact moment when the DBSystem function is first executed. The database value returned by this function remains valid as long as the statement is active. The records in the database have three or more fields as follows:

- The IDKey field – a unique text identifier for the record.
- The Status field – a series of bytes, one for each of the user specified lists, that indicates which lists the record is part of.
- Any other additional fields which have been designated to exist by the user.

For each ID key there is only one record in the database. This is not the case with the transaction log

(discussed later in this topic). In the case of the database that is used to maintain the VTScada alarm system, each alarm has a single record in the database. Although the database is not considered to be a list, it is loosely described as one in such statements as DBListGet, where a value of -1 for the List parameter will use the database rather than one of the user specified lists.

The transaction log is a log file that is saved to disk. For this reason, the larger it becomes the longer it takes to access records from the transaction log. The fields of the transaction log's records are similar to those stored in the database:

- The IDKey field – a unique text identifier for the point associated with the record.
- The Event field – a series of bytes, one for each of the user specified lists, that indicates the event or transaction that has just occurred.
- The Time field – a float value that gives the time of the event, or a user written time value.
- Any other additional fields which have been designated to exist by the user.

Like the database, it is not really considered a list, however in the DBListGet statement, a value of 0 in the List parameter will use the transaction log rather than one of the user specified lists.

The log file differs from the memory database in that for each ID key, there may exist none, one or many records that use the same key. This is because every transaction done by means of the DBTransaction statement is stored in a record in the transaction log.

The lists used by the database system are virtual

lists, which means that they do not exist as actual entities of the system, but rather are composed of the records belonging to the database. The Status field of each record in the database will have exactly the same number of bytes as there are lists. This field will designate which list or lists the record is a member of. In the case of the VTScada alarm system, there are three lists, the active, unacknowledged and disabled alarm lists.

If the user wishes to use this database system simply as a storage device, rather than in a complicated series of lists and transactions like the VTScada alarm system, the NumLists parameter in this function may be set to 0 - no lists will be created. Also, the DBTransFile parameter may be set to invalid, thus disabling transaction logging and causing the transaction log file to not be created.

If the database file has its read-only attribute set when this function is executed, it will be cleared automatically by the function becoming active.

Example:

```
dbSysVal = DBSystem(dbFile { Name of database file },
                    "" { No transaction log },
                    0, 0 { No log, no lists },
                    65 { No. of chars in key },
                    16, 1 { No. of chars in fields 1, 2 },
                    3 { Field 3 is float },
                    1 { Field 4 is short (integer) });
```

DBTrace

Description: This is a trace engine that records live data to a SQL database.

Returns: Numeric

Usage: Steady State only.

Function Groups: Database and Data Source

Related to: See: Trace Viewer Application in the VTScada Programmer's Guide.

Format: DBTrace(FileName, FriendlyName, MyTrigger)

Parameters:

FileName

Required. The name (not the path or extension) of the database file to which DBTrace should record trace data.

FriendlyName

Required. A name for this trace instance that is more meaningful to users.

MyTrigger

Required. The address of the trigger variable that enables this trace.

Comments: DBTrace writes the actual data to the database (using a separate thread) and maintains a live communication buffer with the Trace Viewer.
DBTrace provides a number of methods that allow the trace to define the format of the main table, plus additional tables for filtering and conversion of values to user-meaningful names. Information about these methods are available upon request.

Do not launch this module, rather, launch the constructor (to create the thread).

DBTransaction

Description: Executes in its own thread to perform a transaction on a VTScada database and returns an indication of parameter errors.

Returns: Numeric

Usage: Script Only.

Function Groups: Database and Data Source

Related to: DBAdd | DBInsert | DBGetStream | DBListGet | DBListSize | DBRemove | DBSystem | DBUpdate | DBValue

Threaded: Yes

Format: DBTransaction(DBSysVal, IDKey, Event, EventCode [, Timestamp, FieldVals])
Or
DBTransaction(DBSysVal, IDKey, Event, EventCode [, Timestamp, FieldVal1, FieldVal2, FieldVal3, …])

Parameters:

DBSysVal

Required. The database value to use. This is the return value from a DBSystem call.

IDKey

Required. Any text expression that designates the record to use. It can contain the wildcard characters "*" and "?".

Event

Required. Any text expression containing on byte for each list in the system. Any missing bytes or bytes with values not included in the following list default to "do nothing to list". Note that there is one byte for each list in the system, not for each list that has been selected by the List parameter.

Text Value	Event
+	Add to list
-	Remove from list
<space>	Do nothing to list

EventCode

Required. Any variable in which the event code for the function is returned. This parameter may be used to indicate completion of the function, as it will not be set to a valid value until execution is complete. It has the following meaning

Value	Meaning
0	Record exists and was updated
1	Record did not exist and was added
2	Error occurred

EventCode may be replaced by a constant or Invalid if it is not required.
This parameter will not be set if there is a parameter error (see comments)

Timestamp

Optional. Defines the time to record for the event. If this value is invalid or omitted, the transaction time defaults to the current time. If it is negative, the database file will be updated with the absolute value of this parameter and the transaction log will not be updated.

FieldVals

Required when adding a new record, otherwise, optional. This takes the form of either an array or a list of values. In either case, this is data that is to be written to the database.
When writing to an existing record, the values provided will over-write the values in matching fields.
When creating a new record, all fields must be defined.

Comments: This function executes in the thread created by the DBSystem call, so it will not block other statements from execut-

ing. This does mean, however, that the timing for EventCode becoming valid (marking that the statement has finished executing) is unpredictable and should therefore be checked for validity prior to executing other statements that rely on this statement's results.

The return value for this function indicates if any of its key parameters (DBSysVal, IDKey or Event) are invalid. It will immediately return a value of false (0) unless a key parameter was invalid, in which case it will return true (1). Note that the return value only signals completion of the function's execution if it is true, otherwise the function will continue executing in the thread created for it.

If the user attempts to add a record to a list on which it already exists, that record's time and position on the list will be updated unless the new time is 0, in which case no change will occur.

If the database file has its read-only attribute set when this function is executed, it will be cleared automatically by execution of the function.

Example:

```
db1 = DBSystemDBSystem(dbFile, "", 0, 0, 64 { key }, 10 { field 1 },
1 { field 2 }, -3 { field 3 });
If ValidValid(db1) && ! done;
[
  done = 1;
  DBTransaction(db1 { Database to use },
              "Motor_425" { ID key },
               "--+" { Subtract from lists 1 and 2,
               add to list 3 },
               code { Event code },
               CurrentTime() { Time of the event },
               FieldValues { Array of values for fields });
]
```

DBUpdate

Description: Executes in its own thread to update a VTScada database from a given stream and returns an indication of parameter errors.

Returns: Numeric

Usage: Script Only.

Function Groups: Database and Data Source

Related to: DBAdd | DBInsert | DBListGet | DBGetStream | DBListSize | DBRemove | DBSystem | DBTransaction | DBValue

Threaded: Yes

Format: DBUpdate(DBSysVal, Stream [, EventCode])

Parameters:

DBSysVal

Required. The database value to use. This is the return value from a DBSystem call.

Stream

Required. Any stream value created from the contents of a database. This results from a DBGetStream statement.

EventCode

Required. An optional variable in which the event code for the function is returned. This parameter may be used to indicate completion of the function, as it will not be set to a valid value until finished executing. It has the following meaning

Value	Meaning
0	Database successfully updated
1	Update failed

EventCode may be replaced by a constant or Invalid if it is not required.
This parameter will not be set if there is a parameter error (see comments)

Comments:

This function executes in the thread created by the DBSystem call, so it will not block other statements from executing. This does mean, however, that the timing for EventCode becoming valid (marking that the statement has finished executing) is unpredictable and should therefore be checked for validity prior to executing other statements that rely on this statement's results.

The return value for this function indicates if any of its key parameters (DBSysVal or Stream) are invalid. It will immediately return a value of false (0) unless a key parameter was invalid, in which case it will return true (1). Note that the return value only signals completion of the function's execution if it is true, otherwise the function will continue executing in the thread created for it.

Since the value of the Stream parameter is created from a database, the DBSystem statement that created the database had to have a valid first parameter (the file name in which the database contents are stored). Do not attempt to access the database file directly via a FileStream statement if any DBSystem functions referencing this file are active, since the results may be unpredictable.

This function is useful to synchronize the database setup on two machines over a network. It allows a secondary machine to load the file from the first, thus duplicating its attributes precisely.
If the database file has its read-only attribute set when this function is executed, it will be cleared automatically by execution of the function.

Example:

```
oldDB = DBSystem(oldFile, "", 0, 0, 32 { key }, 2 { field 1 },
                 3 { field 2 });
newDB = DBSystem(newFile, "", 0, 0, 32 { key }, 2 { field 1 },
```

```
                  3 { field 2 });
If Valid(newDB) && ! gotStream;
[
  gotStream = 1;
  DBGetstream(newDB, stream);
]
If Valid(oldDB) && Valid(stream) && ! updated;
[
  updated = 1;
  DBUpdate(oldDB { Database to update },
         stream { Database from which to update },
          code { Event code holder });
]
```

DBValue

Description: Returns a certain value retrieved from a VTScada database.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Database and Data Source

Related to: DBAdd | DBInsert | DBListGet | DBListSize | DBRemove | DBSystem | DBTransaction | DBValue

Format: DBValue(DBSysVal, IDKey, FieldNumber[, ListNum])

Parameters:

DBSysVal

Required. The database value to use. This is the return value from a DBSystem call.

IDKey

Required. Any text expression that designates the record to use. It can contain the wildcard characters "*" and "?".

FieldNumber

Required. Any numeric expression that indicates the field for which data is requested. Valid values for FieldNumber are as follows

Value	Meaning
-1	A timestamp is required. A ListNum parameter must be supplied, specifying which list the timestamp is to be retrieved from.
0	Event code for the record
1 to 255	Indicates which piece of the record's data you wish to retrieve.

ListNum

Optional numeric. When FieldNumber is -1, specifying that a timestamp is required, then that timestamp will be taken from the numbered list in the DB system matching this parameter. The numbered lists start at zero.

See: DBSystem for related information.

Comments:

Unlike most of the other database statements, this function is not threaded. Execution of all other statements will wait until this statement has completed execution and returned its result.

If no records matching IDKey are found, the return value will be Invalid.

If the database file has its read-only attribute set when this function is executed, it will be cleared automatically by execution of the function.

Example:

```
myDB = DBSystem(dbFile, "", 0, 0, 16 { key }, 2 { field 1 },
           3 { field 2 });
If Valid(DBValue(myDB { Database to use },
         "Motor_67" { ID key },
         1 { Field to get value from }) Next;
[
```

```
  ...
]
```

DDE

Description: Returns the value of the data for a specific item from a DDE server program. This function is a DDE client.

Returns: Varies

Usage: Steady State only.

Function Groups: DDE

Related to: DDEPoke | DDEShareAdd | DDEShareDel | SetDDEServer

Format: DDE(Program, Topic, Item[, Error, Trigger, pollTimeOut])

Parameters:

Program

Required. Any text expression giving the name of the program which is the DDE server. This does not contain the .EXE extension. This is usually the same as the root file name of the executable file, but may be different as in the case of Microsoft Word for Windows 6.0 which uses the name "MSWord".
For NetDDE, the program name is of the form "\\Computer\NDDE$" where "Computer" is the name of the computer where the DDE server program is running.

Topic

Required. Any text expression giving the DDE topic name within the server. For a VTScada server the topic name is usually the name of the window. For Microsoft Excel, the topic is the spreadsheet name.
For NetDDE, the topic name is the DDE share name set up in the Windows [ddeshares] section of the SYSTEM.INI file. This configuration section relates a network share name to a program name and individual topic in the DDE server. The SYSTEM.INI must be configured in the DDE server to enable NetDDE.

Item

Required. Any text expression giving the location or name of the value to retrieve.

Error

Optional. A variable that will be set when an error occurs. Values may be as follows: 0=OK, -1=DDE is stopping, -2=FAILED

Trigger

Optional. If present, this indicates a "COLD" link as opposed to a "HOT" link (i.e. you have to poll the server to get an update, rather than it pushing updates to you).
When set to 1, this triggers a poll. The trigger is then reset.

pollTimeOut

Optional. If this is a COLD link (see Trigger) then pollTimeOut is the value in seconds of the time out for the poll.

Comments: This sets up a DDE client. Once the link is established, the return value of the function will change whenever the value in the DDE server changes.

Example:

```
cellValue = DDE("Excel", "Sheet1", "R1C1");
```

Upon execution of this statement, cellValue will contain the value of the cell in row 1, column 1 of the Excel spreadsheet called "Sheet1".

DDEPoke

Description: Sends a value for a specific item to a DDE server program.

Returns: Boolean

Usage: Steady State only.

Function Groups: DDE

Related to: DDE | DDEShareAdd | DDEShareDel | SetDDEServer

Format: DDEPoke(Program, Topic, Item, Value)

Parameters:

Program

Required. Any text expression giving the name of the program which is the DDE server. This does not contain the .EXE extension. This is usually the same as the root file name of the executable file, but may be different as in the case of Microsoft Word for Windows 6.0 which uses the name "MSWord".
For NetDDE, the program name is of the form "\\Computer\NDDE$" where "Computer" is the name of the computer where the DDE server program is running.

Topic

Required. Any text expression giving the DDE topic name within the server. For a VTScada server the topic name is usually the name of the window. For Microsoft Excel, the topic is the spreadsheet name.
For NetDDE, the topic name is the DDE share name set up in the Windows [ddeshares] section of the SYSTEM.INI file. This configuration section relates a network share name to a program name and individual topic in the DDE server. The SYSTEM.INI must be configured in the DDE server to enable NetDDE.

Item

Required. Any text expression giving the name or location of the value to send.

Value

Required. Any expression for the value to be sent. If this value is invalid, a null value (null text string) will be sent to the DDE server program.

Comments: Once the link is established, new data will be sent to the server whenever Value changes. The return value is true (1) if successful and false (0) otherwise.

Example:

```
success = DDEPoke("Excel", "Sheet1", "R1C1", "Connected");
```

Upon successful execution of this statement, the Excel spreadsheet called "Sheet1" will contain the word Connected in row 1, column 1, and success will have a value of 1. If the DDEPoke is not successful, success will be 0 and the Excel cell will retain its previous contents.

DDEShareAdd

Description: Adds a new DDE share name to the SYSTEM.INI file or the registry and returns its own error code.

Returns: Numeric

Usage: Script Only.

Function Groups: DDE

Related to: DDE | DDEPoke | DDEShareDel | SetDDEServer

Format: DDEShareAdd(Object, ShareName, Password)

Parameters:

Object

Required. Any object expression giving the module instance whose variables' value will be shared.

ShareName

Required. Any text expression that gives the DDE share name other DDE enabled programs use to access the variables in Object.

Password

Required. Any text expression that gives the password required to access this DDE share name. If this is

invalid, no password is required.

Comments: The return value of this function is true if successful, false if unsuccessful, and invalid if not attempted (i.e. invalid parameters).

DDEShareDel

Description Deletes a DDE share name from the SYSTEM.ini file or the registry and returns its own error code.

Returns Boolean

Usage Script Only.

Function Groups DDE

Related to: DDE | DDEPoke | DDEShareAdd | SetDDEServer

Format DDEShareDel(ShareName)

Parameters

ShareName

Required. Any text expression that gives the DDE share name.

Comments The return value of this function is true if successful, false if unsuccessful, and invalid if not attempted (i.e. invalid parameters).

DeadBand

Description: Returns the previous value of the first parameter until it changes by an amount specified by the second parameter.

Returns: Numeric

Usage: Steady State only.

Function Groups: Rounding Math

Related to: Change | PID

Format: Deadband(Value, Delta)

Parameters:

Value

Required. Any numeric expression giving the value whose changes are to be limited to amounts no less than Delta.

Delta

Required. Any numeric expression giving the lower limit of the change in Value that will allow the function value to change. If Delta is less than or equal to 0, the function will always return Value.

Comments: When the function is first called, its value will be that of its first parameter Value. After that, the value will remain at this initial value until the difference between Value and the function value is greater than or equal to Delta. At this point, the function value will assume the new value of Value. This has the effect of ignoring all changes in Value which are less than Delta. This is useful for ignoring "noise" in values such as analog inputs. Since VTScada performs calculations based upon changes in values, this function may be used in certain situations to minimize the number of irrelevant value changes and thereby reduce the frequency of calculations and improve performance. This function may also be used by PID to eliminate noise in the process value parameter and therefore avoid making frequent minor output changes which are irrelevant.

The change specified by Delta is the absolute value of the change so a positive or negative change in Value will be treated equally. Delta should be always greater than zero.

Example:

```
ztext(500, 100 { X, Y coordinates },
      Format(0, 2, DeadBand(pressure, 5)) { Displayed value },
      0, 0);
```

This displays the value of variable, "pressure" on the screen. However, the display is only updated when pressure changes by more than 5 (higher or lower) from the last update.

Related Information:

See: "Latching and Resetting Functions" in the VTScada Programmer's Guide

Debugger

(System Library)

Description: Starts the VTScada debugger.

Returns: Nothing

Usage: Script or steady state.

Function Groups: Compilation and On-Line Modifications

Related to:

Threaded: Yes

Format: \System\Debugger([Status, ViewModule, TxtAppName, MtSdView, Restrict])

Parameters:

Status

Optional. Any numeric expression giving the enabled status of the window in which the debugger is displayed, as follows. The default value is 1 - Enabled.

Status	Meaning
0	Disabled
1	Enabled
2	Enabled, bring to front
3	Enabled, reload ViewModule, then bring to front

ViewModule

Optional. An object value selecting the current module or object to debug. No default value.

TxtAppName

Optional. Any text expression to be appended to the title bar of the debugger window. No default value.

MtSdView

Optional. Any logical expression. If true (non-0) the module tree and state diagram buttons are displayed on the debugger, if false (0), neither will be displayed. The default is false.

Restrict

Optional. Any logical expression. If true (non-0) VTScada system windows may not be selected for debugging, if false (0) any window may be selected by the debugger, including itself. The default is true.

Comments: This module is a member of the System Library, and must therefore be prefaced by \System\, as shown in the "Format" section. If you are developing a script application, use "System\..." rather than "\System\..." in the function call.
For any optional parameter that is to be set, all optional parameters preceding the desired one must be present, although they may be invalid.

Decode

(System Library)

Description: Returns the plain value of a cipher that is the result of the Encode function.

Returns: String

Usage: Script Only.

Function Groups: Encryption

Related to: Encode | BlockDecrypt | Base64Decode | Hash | Unpack

Format: \System\Decode(CipherValue[, UnpackDictionary, Key, HashKey);

Parameters:

CipherValue

Required. The information to be decoded.

UnpackDictionary

If a dictionary was used to pack the information as part of the encoding, the mirror of that dictionary must be provided to unpack the information.
See notes and example in the Unpack function.

Key

Optional string. Must be included if the CipherValue was encrypted using a key, in which case this value must match the key that was used in the Encode function.

HashKey

Optional string. Must be included if the CipherValue was protected from tampering with a Hash value, in which case this value must match the Hash that was used in the Encode function.

Comments: none

Examples:

Decommission

(Alarm Manager module)

Description: Decommission an alarm by name.

Returns: Nothing

Usage: Script Only.

Function Groups: Alarm

Related to: Commission

Format: \AlarmManager\Decommission(AlarmName)

Parameters:

AlarmName

Required text. The alarm name. Typically, the unique id of the alarm tag, or the tag containing built-in alarms.

Comments: An alarm tag that is deleted without being decommissioned is referred to as an orphaned alarm. VTScada will removed orphaned alarms automatically, but it is better practice to specifically decommission the alarm

Example:

The following would typically be found in a tag's Refresh state.

```
IfElse(Valid(Name), Execute(
                      { ... Alarm commissioning code ... }
                    );
      { Else }
      IfThen(Valid(Parm[#Name]),
             \AlarmManager\Decommission(Root\UniqueID);
           );
);
```

Decrypt

Description: The Decrypt function decrypts data previously encrypted using the Encrypt function. It is the VTScada analog of the CryptoAPI CryptDecrypt call.

Returns: Text

Usage: Script Only.

Function Groups: Cryptography

Related to: DeriveKey | Encrypt | ExportKey | GenerateKey | GetCryptoProvider | GetKeyParam | ImportKey | SetKeyParam

Format: Decrypt(Key, CipherText, Final [, Reserved, Flags, Error])

Parameters:

Key

Required. The handle to the key to use to decrypt the data.

CipherText

Required. A text string that contains the cipher text to be decrypted.

Final

Required. A parameter that specifies whether this is the last section in a series being decrypted. Final is set TRUE for the last or only block and FALSE if there are more blocks to be decrypted

Reserved n/a

An optional parameter which should be set to 0. If omitted or invalid, then the value 0 is used.

Flags

Optional. An optional parameter specifying the flags to be passed to CryptDecrypt. If omitted or invalid then the value 0 is used.

Error

Required. An optional variable in which the error code for the function is returned. It can have the following meanings:

Value	Meaning
0	Key successfully imported.
1	Key, CipherText or Final parameters invalid.
x	Any other value is an error from CryptDecrypt.

Comments: The plain text is returned as a text string. If an error occurs, the return value is invalid.

Example:

```
[
  PlainText2;
]
Init [
  If 1 Main;
  [
    PlainText2 = Decrypt(Key4, CipherText1, 1, 0, 0);
  ]
]
```

DefaultNamingContext

Description: Returns the string value of the LDAP default naming context for the host machine domain.

Returns: Text

Usage: Steady State only.

Function Groups: Security

Related to:

Format: DefaultNamingContext()

Parameters: None

Comments: If the host machine is unable to contact the domain controller invalid is returned, otherwise a string of the form "dc=example,dc=com" is returned
This function is useful only for Active Directory operations.
Note that this function may take some time to complete and will block the caller until it does. Other threads in the VTScada system will continue to run while this function is executing.

DefaultPrinter

Description: Returns the Windows™ default printer.

Returns: Handle

Usage: Script Only.

Function Groups: Printer

Related to: Redirect

Format: DefaultPrinter()

Parameters: None

Comments: This function is used primarily in conjunction with the Redirect statement, which permanently redirects the resources of the system. The only way to return to the original default printer without knowing its source in advance is by using this function as illustrated in the "Example" section.

Example:

```
If MatchKeys(2, "p") { When letter "p" is pressed };
[
  str1 = DefaultPrinter() { Store the default printer };
  Redirect("DEF:", newPrinter) { Select different printer };
  PrtScrn() { Output screen to printer };
  Redirect("DEF:", str1) { Restore original printer };
]
```

In this script, the default printer is saved before redirecting printing to another printer. This allows the original printer to be restored as the default when the custom printing is concluded.

Deflate

Description: Compresses/decompresses a buffer of data using the DEFLATE algorithm, and returns the compressed/decompressed data.

Returns: Numeric

Usage: Script Only.

Function Groups: String and Buffer

Related to:

Format: Deflate(InputBuffer, Decompress [, DeflateHandle, MaxDecompressedLength, FlushMode, CompressionLevel, Format])

Parameters:

InputBuffer

Required. Any expression that will yield a buffer of data to be compressed or decompressed. This can be a string, a buffer, or a stream.

Decompress

Required. A Boolean value. If set to true (non-zero), the data in the InputBuffer is decompressed. If set to false (zero), the data in InputBuffer is compressed.

DeflateHandle

Optional. A variable that holds an existing deflate handle or is to receive a new deflate handle (see the "Comments" section for further details).

MaxDecompressedLength

Optional. Specifies the maximum length of the decompressed output (see the "Comments" section for further details).

FlushMode

Optional. Indicates the type of flushing that the encoder should use when processing the data. Values

FlushMode	ZLib Value
0	Z_NO_FLUSH
2	Z_SYNC_FLUSH
3	Z_FULL_FLUSH
4	Z_FINISH

Effective use of this parameter requires an understanding of the ZLib library (www.g-zip.org/zlib). The default value is Z_SYNC_FLUSH which is appropriate for continuous, stream mode type operations. For a single encoding, as in a file or a web page, then Z_FINISH is likely to be more appropriate. It is not normally necessary to specify a flush value for decompress but, if one is specified, it will be used.

CompressionLevel

Optional. Indicates the level of aggressiveness of the encoder. Values range between 0 and 9 with typical values being

CompressionLevel	ZLib Value
0	Z_NO_COMPRESSION
1	Z_BEST_SPEED
9	Z_BEST_COMPRESSION
-1	Z_DEFAULT_COMPRESSION

The default value used by VTScada is Z_BEST_COMPRESSION which should be fine for most applications. Z_DEFAULT_COMPRESSION requests a default compromise between speed and compression (equivalent to level 6). Again, it is not necessary to specify this on decompression as the decoder can determine the encoding scheme used.

Format

Optional. Indicates the type of compression to use. ZLib is more intended for streams, while GZip for files. Both are simply wrappers around the actual deflated data. Raw Deflate can be used to obtain or process just the deflated data without the ZLib or GZip overhead. Values are

Format	Compression Type
0	ZLib (default)
1	GZip
2	Raw Deflate

Comments:

Compression/decompression is performed using an implementation of the DEFLATE algorithm (RFC 1951).
The DEFLATE algorithm is a dictionary-based compression technique, suitable for many types of data. A dictionary-based algorithm needs to include dictionary tokens in the output buffer, so compression of small buffers can give poor results.

The DeflateHandle parameter is an opaque quantity that identifies an existing dictionary, or, if the variable it identifies does not hold an existing deflate handle, is set to a new deflate handle on return from the statement. Supplying the same deflate handle for successive calls to Deflate results in the existing dictionary being used and augmented on successive calls. This usually improves compression significantly.

If you make a series of calls to Deflate, using the same deflate handle to decompress, then you must make a series of calls in the same sequence, with the initial call obtaining a new deflate handle, and subsequent calls using the newly-acquired deflate handle (see the "Example" section for further details).

MaxDecompressedLength is useful when you already know the size of the decompressed data (usually because you recorded it before compression was performed). While this parameter is not necessary, its use reduces the memory required to decompress the data.

The default size of MaxDecompressedLength is 65536 bytes. Expanding data larger than this size will cause the output to be clipped unless a larger value is provided in this parameter.

Example:

```
If 1 Done;
[
  CompressedData = Deflate("Mary had a little lamb", 0{compress});
  OriginalData = Deflate(CompressedData, 1{decompress});
]
```

In the above example, OriginalData will hold "Mary had a little lamb", after the second call to Deflate.

```
If 1 Done;
[
  CompressHandle = DecompressHandle = Invalid;
  CompressedData = Deflate("Mary had a little lamb", 0{compress},
                           CompressHandle);
  OriginalData = Deflate(CompressedData, 1{decompress},
                         DecompressHandle);
  CompressedData = Deflate("Mary had a little lamb", 0{compress},
                           CompressHandle);
  OriginalData = Deflate(CompressedData, 1{decompress},
                         DecompressHandle);
  CompressedData = Deflate("Mary had a little lamb", 0{compress},
                         CompressHandle);
  OriginalData = Deflate(CompressedData, 1{decompress},
                         DecompressHandle);
]
```

In this example, OriginalData will still hold "Mary had a little lamb" after each call to Deflate, however successive use of the same dictionary will result in later compressions of the string being more efficient (i.e. smaller). This is because the second and third compressions do not need to send dictionary tokens in the output stream, as the tokens are "remembered" in DecompressHandle. This technique allows highly

efficient "stream-oriented" continuous compression, so long as the compressed data is fed into Deflate in exactly the same order as it was compressed. Note that separate handles need to be used for both compression and decompression, and that these handles will both be initialized on the first call to Deflate in which they are used.

DeleteAccount

Security Manager Module

Description Removes an account.

Returns Object value

Usage Script Only.

Related to: AddAccount | ModifyAccount

Format \SecurityManager\DeleteAccount (NewAccountData [, PtrReturnCode, HaveLock]);

Parameters

NewAccountData

Required. An AccountData structure, a single dimension array of AccountData structures or a dictionary of AccountData structures identifying the account(s) to delete.

PtrReturnCode

Optional. A pointer to a value that will contain one of the defined result codes at the conclusion of the operation.

HaveLock

Optional. A Boolean value that indicates whether the working copy lock is held by the calling code. Default FALSE.

Comments To use this API, the calling code must be running in a secur-

ity session that has Manager privilege.
Deleting an account is an asynchronous operation. If the asynchronous operation was not attempted, due to detection of an error, the return value will be Invalid. If the asynchronous operation is attempted, the return value will be an object value. The object value will become Invalid when the asynchronous operation completes. At that time (or when the method returns Invalid), the value addressed by PtrReturnCode can be examined to determine the status of the operation. The contents of the value addressed by PtrReturnCode is undefined until the method returns Invalid.
A single account can be deleted by supplying a single AccountData structure in NewAccountData. Multiple accounts can be deleted in one operation by providing a single dimension array or dictionary of AccountData structures in NewAccountData.
The result code returned in the value addressed by PtrReturnCode will be a scalar value if a single structure was supplied in NewAccountData. If an array of structures or a dictionary of structures was supplied, a single dimension array of the same size as NewAccountData will be returned in the value addressed by PtrReturnCode, each element containing the result code for the corresponding NewAccountData element.
Deleting an account requires a working copy write lock. If such a lock is held by the calling code, the HaveLock parameter must be set to TRUE. Otherwise omit this parameter or set it to FALSE. If the calling code holds a read lock on the working copy, this must be released before DeleteAccount can complete its operation.
The AccountData structure(s) provided must have the AccountID member set to an existing account ID. All other members are ignored.

DeleteArrayItem

Description: Deletes an element from a single-dimension, dynamically-allocated array and returns the modified array.

Returns: Array

Usage: Script Only.

Function Groups: Array

Related to: InsertArrayItem | New

Format: DeleteArrayItem(Array, Index)

Parameters:

Array

Required. Any array variable that has been created via a New function call. This should be a single dimension array or unexpected results may occur.

Index

Required. Any numeric expression giving the element in the array to delete. If out of range, then the original array will be returned.

Comments: This function supersedes the System Library's DeleteListItem.
This function is intended for use on dynamically allocated arrays, that is, arrays that have been created via the New function. If used with an array that has been statically declared, unless otherwise specified in the Array parameter, the first element of the array will be used. If this element contains a dynamically allocated array, the deletion of the specified element will be performed on this array.

Example:

```
If 1 Next;
[
  Data = DeleteArrayItem(Data, ArraySize(Data, 0) - 1);
]
```

This statement deletes the last element in the array called Data.

DeleteContributor

Description: Removes a contributor from a container.

Returns: Numeric

Usage: Script Only.

Function Groups: Containers and Contributor

Related to: AddContributor | GetContributors | PContributor

Format: DeleteContributor(HandleName, ArrayName, CountName, ContainerObj, ContributorObj, CountIncrement);

Parameters:

HandleName

Required. The name of the handle variable in the container module.

ArrayName

Required. The name of the variable in the ContainerObj that holds an array of values from which to delete the contributor. This parameter may be invalid if there is no such array in the container.

CountName

Required. The name of the variable in the ContainerObj that holds a count of the current number of this type of contributor. This parameter may be invalid if no such variable exists in the ContainerObj. Not all contributors need to be counted. The CountIncrement determines the initial change in the count and the contributor must maintain the count.

ContainerObj

Required. The object value of the container tag module.

ContributorObj

Required. The object value of the contributor to delete.

CountIncrement

Required. This value will be subtracted from the variable in the container that has the name of CountName. This value is usually a "1" or a "0", indicating whether or not the contributor is actively contributing its value now. The contributor will increment or decrement the value of the CountName variable as the corresponding state of the contributor changes.

Comments: This function can be called from the contributor.

DeleteModule

Description: Deletes a module from the system.

Warning: This function should be used by advanced users only. Irrevocable alteration of your application may occur

Returns: Numeric (see table in comments)

Usage: Script Only.

Function Groups: Compilation and On-Line Modifications, Advanced Module

Related to: ClearModule | DeleteOptional | DeleteState | DeleteStatement | DeleteVariable

Format: DeleteModule(Module[, IgnoreSource])

Parameters:

Module

Required. Any expression that gives the module value of the module to delete.

IgnoreSource

If true, the function will not modify any source file. Also, will not check that .SRC and .RUN files are in sync

and therefore will not return a value of 3.
Defaults to FALSE if invalid or not specified.

Comments: This statement deletes a module from the system, as well as removing its code from the document file, if certain conditions hold. There must not be any running instances of this module. All of the variables in the module must only be referenced within that module. That is, none of the variables can be referenced outside of the module which they are in. Module itself must not be referenced in any other module at all. This function returns values as follows

Return Value	Meaning
0	Module successfully deleted.
1	Module not deleted, there are instances of it running.
2	Module not deleted, module is externally referenced.
3	Module not deleted, run files are out of sync.

DeleteOptional

Description: Deletes a statement from an action script.

Warning: This function should be used by advanced users only. Irrevocable alteration of your application may occur.

Returns: Nothing

Usage: Script Only.

Function Groups: Compilation and On-Line Modifications, States

Related to: DeleteModule | DeleteState | DeleteStatement | DeleteVariable

Format: DeleteOptional(Action, Position)

Parameters:

Action

Required. Any expression that gives the VTScada Value Types – Numeric Reference of the action.

Position

Required. Any numeric expression giving the statement number in the script to delete, beginning with 1.

Comments: The corresponding text for the deleted statement is removed from the document file.
This statement is disabled in the run time version of VTScada. It will do nothing.

DeletePrivFromUser

(Security Manager Library)

Description: Removes a privilege from the specified username.

Returns: Numeric (via the first parameter)

Usage: Script Only.

Function Groups: Security

Related to:

Format: \SecurityManager\DeletePrivFromUser(PtrReturnCode, Username, Privilege[, HaveLock])

Parameters:

PtrReturnCode

Required. A pointer to a variable that will be used for the return code.

PtrReturnCode	Meaning
1	Privilege deleted.
2	Denied. The calling context does not have the Manager system privilege.
3	The privilege is not valid – no action taken.
4	The specified user does not exist – no action taken.
6	The application cannot be edited.

UserName

Required. Any expression for the name of the user account to modify.

Privilege

Required. Any numeric expression for the privilege to be denied. Use a negative value for a system privilege and a positive value for an application privilege.

HaveLock

Optional Boolean expression. Set to true if we have the WC lock. Defaults to 0 or FALSE.

Comments:

May only be called from a user-context that has the Manager system privilege. The return value of the function is the object value of the launched worker module. This will be set to Invalid when the operation has completed and may be used to discover when that occurs.
Use of this function requires an understanding of the VTScada security system and the system privileges. Please refer to System Privileges in the chapter Security Manager

Service.

DeleteState

Description: Deletes a state from a module.

Warning: This function should be used by advanced users only. Irrevocable alteration of your application may occur.

Returns: Nothing

Usage: Script Only.

Function Groups: Compilation and On-Line Modifications, States

Related to: ClearState | DeleteModule | DeleteOptional | DeleteStatement | DeleteVariable

Format: DeleteState(State)

Parameters:

State

Required. Any expression that gives the code value of the state.

Comments: The corresponding text for the deleted state is removed from the document file.

DeleteStatement

Description: Deletes a statement from a state.

Warning: This function should be used by advanced users only. Irrevocable alteration of your application may occur.

Returns: Nothing

Usage: Script Only.

Function Groups: Compilation and On-Line Modifications, States

Related to: DeleteModule | DeleteOptional | DeleteState | DeleteVariable

Format: DeleteStatement(Statement)

Parameters:

Statement

Required. Any expression that gives the code value of the statement.

Comments: The corresponding text for the deleted statement is removed from the document file.
This statement is disabled in the run time version of VTScada. It will do nothing.

DeleteUser

(Security Manager Library)

Description: Removes a specified user name from the system.

Returns: Numeric (via the first parameter)

Usage: Script Only.

Function Groups: Security

Related to:

Format: \SecurityManager\DeleteUser(PtrReturnCode, Username[, HaveLock])

Parameters:

PtrReturnCode

Required. A pointer to a variable that will be used for the return code.

PtrReturnCode	Meaning
0	User does not exist. No action taken.
1	User deleted.
2	Denied. The calling context does not have the Manager system privilege.
6	The application cannot be edited.

UserName:

Required. Any expression for the name of the user account to delete.

HaveLock:

Optional Boolean expression. Set to true if we have the WC lock. Defaults to 0 or FALSE.

Comments: May only be called from a user-context that has the Manager system privilege. The return value of the function is the object value of the launched worker module. This will be set to Invalid when the operation has completed and may be used to discover when that occurs.
Use of this function requires an understanding of the VTScada security system and the system privileges. Please refer to System Privileges in the chapter Security Manager Service.

DeleteVariable

Description: Deletes a variable from a module.

Warning: This function should be used by advanced users only. Irrevocable alteration of your application may occur.

Returns: Nothing

Usage: Script Only.

Function Groups: Compilation and On-Line Modifications, Variable

Related to: DeleteModule| DeleteOptional| DeleteState | DeleteStatement

Format: DeleteVariable(Variable[, IgnoreSource])

Parameters:

Variable

Required. Any expression that gives the variable value to delete.

IgnoreSource

An optional Boolean expression. If true, the function will ignore out-of-sync source files and not attempt to make any source file changes. The result is convenient deletion of non-temporary variables without having the function modify the corresponding source code. Defaults to FALSE.

Comments: If IgnoreSource is FALSE or not provided then the corresponding text for the deleted variable is removed from the source file. However, this is only true if the variable is not a temporary variable, and only if the files are in sync. DeleteVariable will fail if there are any references to the variable to be deleted.

DelPageFromApp

Description: Deletes a system page from an application.

Warning: This function should be used by advanced users only. Irrevocable alteration of your application may occur.

Returns: Nothing

Usage: Script Only.

Function Groups: Compilation and On-Line Modifications, Window

Related to:

Format: \DelPageFromApp(PageName);

Parameters:

PageName

Required. The text name of the page to delete.

Comments: In versions of VTS prior to 8.1, this subroutine was a part of the page manager and therefore was called using the format \PageManager\DelPageFromApp(...). With version 8.1, the page manager is no longer a part of VTScada and the function can simply be called using \DelPageFromApp () as shown in the "Format" section above.

DelRead

Description: Is called by a tag to delete an existing read request, as created by an AddRead.

Returns: Nothing

Usage: Script Only.

Function Groups: Memory I/O

Related to: AddRead

Format: VTSDriver\DelRead(Address, Value, Rate)

Parameters:

Address

Required. The address from which to get the data.

Value

Required. A pointer to the destination for the read data.

Rate

Required. The update rate in seconds.

Comments: DelRead() can only delete a single item. The module searches ReadList (a linked list of ReadBlocks) for the node whose info vars match those of the item to be deleted. It then looks for the matching rate, then the specific request as identified by Value (the address of the tag's RawValue) and removes it from the list.

Deriv

Description: Returns the derivative (rate of change) of a value.

Returns: Numeric

Usage: Steady State only. See: Rules for Usage.

Function Groups: Generic Math

Related to: Intgr | PID

Format: Deriv(Value, Time)

Parameters:

Value

Required. Normally, the name of a variable holding a numeric value for which the derivative will be taken.

Time

Required. Any numeric expression giving the maximum time in seconds between derivative function updates.

Comments: This function takes the change in the Value parameter between two successive evaluations and divides by the elapsed time interval between these evaluations. If the Value changes from invalid to valid, it will take two evaluations before the function's result becomes valid. This function is the inverse of Intgr.
The time parameter is necessary because of VTScada's evaluation method of not doing any calculations unless necessitated by a change in a parameter. This means that if the

Value remains unchanged, the Deriv function will be recalculated after the time interval specified by the Time parameter and return 0.
This function is often used in control functions such as PID loops where it makes up the "D" in the "PID".

Example:

```
LevelROC = Deriv(Level, 0.1);
```

Assuming that Level is updated to show a changing value, this will set LevelROC to the rate of change of that level.

DeriveKey

Description: Generates a cryptographic session key from a seed value.

Returns: Handle

Usage: Script Only.

Function Groups: Cryptography

Related to: Decrypt | Encrypt | ExportKey | GenerateKey | GetKeyParam | ImportKey | SetKeyParam

Format: DeriveKey(CSPHandle, AlgID, Seed [, Flags, Error])

Parameters:

CSPHandle

Required. The handle of a CSP to use to generate the key.

AlgID

Required. Identifies the algorithm for which the key is to be generated. Values for this parameter vary depending on the CSP used, and are defined in WinCrypt.h

Seed

Required. A text string to use as a seed.

Flags

Optional. Specifies the flags to be passed to CryptGenKey. If omitted or invalid, then the value "0" is used.

Error

Optional. A variable in which the error code for the function is returned. The error codes are as follows

Error	Meaning
0	Key successfully generated
1	CSPHandle or AlgID parameters invalid
X	Any other value is an error from CryptGenKey.

Comments:

DeriveKey guarantees that when the same CSP and algorithms are used, the keys generated from the same seed are identical. The base data can be a password or any other user data. A handle to the key or key pair is returned. This handle can then be used as needed with any Crypto API function requiring a key handle. It is the VTScada analog of the Crypto API CryptDeriveKey call.

The return value for this function is a handle to the Key. If an error occurs, then the return value is Invalid. A key has a value type of 37. If cast to text, then the hexadecimal value of the algorithm ID will be returned.

Example:

```
[
  Key1;
  Constant CALG_RC4 = 0x6801;
  Constant KEY_SIZE = 40;
  Constant Password = "A secret password";
]
Init [
  If 1 Main;
  [
    { Make a key }
    Key1 = DeriveKey(CSP, CALG_RC4, Password, KEY_SIZE << 16);
```

```
  ]
]
```

DialogInitPos

(System Library)

Description: Attempts to position a dialog so that it is not started beyond the left, right, top, or bottom of the visible screen.

Returns: Nothing

Usage: Script Only.

Function Groups: Graphics

Related to:

Format: \System\DialogInitPos(XPosPtr, YPosPtr [, DefaultXPos, DefaultYPos, DlgWidth, DlgHeight])

Parameters:

XPosPtr

Required. A pointer to the dialog's X-position variable. The X-position will be modified by this function if the dialog is beyond the left or right edge of the screen.

YPosPtr

Required. A pointer to the dialog's Y-position variable. The Y-position will be modified by this function if the dialog is beyond the top or bottom edge of the screen.

DefaultXPos

Optional. Any numeric expression giving the default X-position to use if the value of the XPosPtr is Invalid. It is also the X-position that will be used if the dialog is beyond the right edge of the screen and Width is Invalid. Default is 0.

DefaultYPos

Optional. Any numeric expression giving the default Y-position to use if the value of the YPosPtr is Invalid. It is also the Y-position that will be used if the dialog is

beyond the bottom edge of the screen and Height is Invalid. Default is 0.

Width

Optional. Any numeric expression giving the width of the dialog.

Height

Optional. Any numeric expression giving the height of the dialog.

Comments:

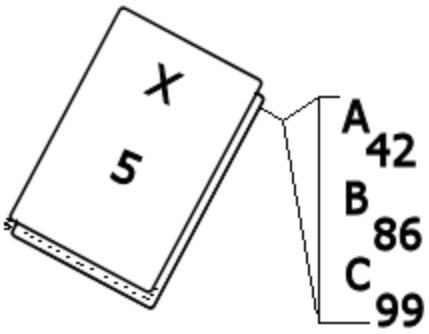
This module is a member of the System Library, and must therefore be prefaced by \System\, as shown in the "Format" section. If you are developing a script application, use "System\..." rather than "\System\..." in the function call.

Calling this function before starting a Window() call will reposition a dialog's starting position so it will not be drawn entirely off-screen. This is only necessary if the dialog's X- or Y-coordinates are retained. The width and height of the dialog are optional. If provided, the dialog will be prevented from being drawn partially beyond the bottom/right edge of the screen. Otherwise only the top/left corner of the dialog will be guaranteed to be drawn on-screen. There is no return value.

Dictionary

Description:

Creates a database-like storage structure that provides efficient addition, retrieval and removal of information linked to key values.
Keys can be any data type although integers and strings are recommended. Values can be any data type including another dictionary.



Returns: Dictionary

Usage: Script Only.

Function Groups: Dictionary, Variable

Related to: MetaData | DictionaryCopy | DictionaryRemove | GetNextKey | GetKeyCount | HasMetaData | IsDictionary | ListKeys | RootValue

Format: Dictionary([case] , [root]);

Parameters:

Case

Optional. A Boolean indicating whether the keys in the dictionary are to be case sensitive.
TRUE == Not Case Sensitive (default)
FALSE == Case Sensitive

Root

Optional text value. Numeric values will be cast to text. If not provided, the dictionary will have no root value. (default: Invalid)

Example:

```
X = Dictionary(0, 5);
X["A"] = 42;
X["B"] = 86;
X["C"] = 99;
```

DictionaryCopy

Description: Create a new dictionary with contents identical to an existing dictionary. It is expectd that this function will be used rarely, since in most cases it will be more efficient to hand off a reference to a dictionary rather than build a duplicate of it.
In the case of a complex dictionary that contains other dictionaries within it, the optional Boolean parameter, deep, controls whether the copy should also contain the sub dictionaries, as does the original, or if it should contain copies of those dictionaries.

Returns: Dictionary

Usage: Script Only.

Function Groups: Dictionary, Variable

Related to: Dictionary | MetaData | DictionaryRemove | GetNextKey | GetKeyCount | HasMetaData | IsDictionary | ListKeys | RootValue

Format: NewDictionary = DictionaryCopy(dictionary[, deep, acyclic, lock]);

Parameters:

Dictionary

Required. The name of the dictionary.

Deep

Optional. A Boolean that causes all linked dictionaries, if any, to be copied as well. Defaults to FALSE

Acyclic

Optional. A Boolean indicating that cyclic links within the dictionary structure should not be included in the copy. Defaults to FALSE

Lock

Optional. A Boolean indicating that the contents of the

dictionary copy are to be constant. Defaults to FALSE

DictionaryRemove

Description: Removes a key / value pair from a dictionary, providing a means to regain memory space and remove data that is no longer needed.

Returns: Nothing

Usage: Script Only.

Function Groups: Dictionary

Related to: Dictionary | MetaData | DictionaryCopy | GetNextKey | GetKeyCount | HasMetaData | IsDictionary | ListKeys | RootValue

Format: DictionaryRemove (dictionary, key);

Parameters:

Dictionary

Required. The name of the dictionary.

Key

Required. The name of the key to be removed.

Diff

Description: Compares two buffers and generates a third buffer containing formatted instructions describing how the first buffer can be modified so that it will match the second. This will perform a delimited difference unless the ChunkSize parameter is set to 1 or greater.

Returns: Invalid (result returned in second parameter)

Usage: Script Only.

Function Groups: String and Buffer

Related to:

Threaded: Yes

Format: Diff(ResultBuffer, CompletionCounter, Buffer1, Buffer2, [Delimiter, Chunk Size, Clip Length, Edge Length, MaxVariance, PointCap])

Parameters:

ResultBuffer

Required. Any expression that resolves to the variable to be set to the output buffer. This buffer is created asynchronously and should be checked for valid data before use.
The content of this buffer will be an instruction set for transforming the contents of Buffer1 into a duplicate of the contents of Buffer2. A detailed description of this instruction set is provided in the Comments section.

Completion Counter

Required. Any expression that resolves to a variable containing a numeric value or Invalid.
If a numeric variable, the value will be incremented at the instant that Diff is called. It will then be decremented after the Result Buffer has been populated. The same variable can be used to monitor any number of simultaneous, asynchronous Diff operations.
If this parameter is set to Invalid then the Diff operation will be performed synchronously. The function won't return until the Result Buffer is populated.

Buffer1

Required. Any expression that returns the first buffer. This is the buffer that is intended to be modified by the instructions returned.

Buffer2

Required. Any expression for the second buffer. This is the buffer that the first buffer would resemble were the returned instructions applied.

Delimiter

Optional. The bytes used to delimit lines in text buffers (or records in any sort of delimited buffer). Multiple delimiters can be specified by passing an array of text strings.

If not otherwise specified, the default is an array containing typical text file line endings (newline, carriage return or a combination of the two characters in either order).

Can accept either a single string or an array of strings

ChunkSize

Optional. The number of bytes to compare as a unit in a binary buffer. Must be set to 1 or greater to enable a binary diff (a delimited diff is performed by default). Unless the contents of the buffers are guaranteed to align to a given number of bytes it is recommended that this be set to 1 to enable binary diffs. Defaults to 0.

ClipLength

Optional. This numeric value is an optimization. It indicates how long a string of matches (i.e. both buffers having identical contents) will become before the function decides that it has found an optimal instruction set and will discard competing sets.

If Diff returns sub-optimal instructions you should increase this value. Lower values will reduce the execution time of the function at the cost of the quality of the output. Higher values increase output quality but decrease speed.

Sub-optimal instruction sets will result if strings of matches having the given length can occur randomly within the two buffers. Defaults to 20.

EdgeLength

Optional. Another numeric optimization, best set to twice the ClipLength. Causes the elimination of instruction sets that are estimated to require at least EdgeLength more instructions than the best set at any point during the search.
Sub-optimal instruction sets will result if the estimate is inaccurate by an amount greater than this value. Lower EdgeLength values will reduce the execution time of the function at the cost of the quality of the output. Higher values will increase output quality but decrease speed. Defaults to 40.

MaxVariance

Optional. Sets a maximum variance, as measured by the number of items changes in the same way. If the DIFF strays from an exact match by MaxVariance by a given number of data adds or deletes, execution will stop.
A mixture of adds and deletes will cancel each other out. When set to a value smaller than the default, files with lots of small modified areas will pass while files with a single modification, larger than this variance, will fail.
Defaults to 1,000,000.

PointCap

Optional. Sets a cap on the number of points that will be searched within the buffers. In effect, this value serves to cause a timeout when comparing extremely large buffers that are almost completely different.
Defaults to 1,000,000,000.

Comments: This function will return Invalid on failure. Otherwise, the return value is a buffer of zero or more binary records. Each record will consist of at least two 32-bit words, containing instructions in the following form.

The highest bit of the first word indicates whether this is a delete instruction or an add instruction. 0 means "delete" while 1 means "add". The remaining 31 bits of the first word (taken as a 31-bit unsigned integer) contain the number of bits to be affected by this operation.
The second word, taken as a 32-bit unsigned integer, indicates the offset of the operation. That is the location of the bytes affected.
If the operation is to add bytes, there will be a binary string following the second word. These are the bytes to be added at the specified location.
Since the diff function uses a searching algorithm, and in particular an incomplete search (that is it tries to find a solution without exploring all of the possibilities), it will at any time only have a partial collection of all the possible solutions to the problem. Each solution is defined as a set of instructions that modify the source buffer, and each of these sets requires a different number of instructions to convert a different region of that buffer. The "best set" is the one that converts the largest portion of the buffer while requiring the fewest changes to it, selected from the solutions that have been discovered so far.
The optimization works by eliminating solutions which appear to be so much worse than the current best set that they are unlikely to recover, as judged by how many more changes they require to convert a similar region. The problem is that a solution which works poorly in one region may perform much better in the others, so the optimization may cause the "real" best set (the one that's optimal for the entire buffer) to be overlooked.
The return value will be an empty buffer if Buffer1 and Buffer2 are identical.

Dir

Description: Performs a search in the given directory and returns an

array of matching file names.

Returns: Array

Usage: Script only.
May be used in optimized Tag Parameter Expressions.

Function Groups: File I/O

Related to: DriveInfo | FileFind | FileDialogBox

Format: Dir(Path, Attributes, Option)

Parameters:

Path

Required. Any text expression that indicates the full path name for the directory to list. The search is non-recursive (i.e. it considers only the specified directory, not any of its sub-directories). Path may include a wild-card, such as "*.DAT". A known path alias for File-Related Functions may be provided in the form, :{KnownPathAlias}.

Attributes

Required. Any numeric expression that gives the attributes to match on each file listed. Files that don't match these attributes won't be listed. The Attribute parameter may be set as one of the following:

Attribute	Bit No.	Meaning
0	-	Files without attributes
8	3	All files (regardless of attributes)

... or, it may be constructed by adding together the numbers from the following table :

Attribute	Bit No.	Meaning
1	0	Read only
2	1	Hidden
4	2	System
16	4	Sub-directory
32	5	Archive

Option

Required. Any numeric expression giving the type of text information to generate. All information is placed in a single text string (which will be stored in an element of the array that is created). The information is written from left to right, with lowest option numbers first.

Option	Bit No.	Description
1	0	Short file name
2	1	Full path and file name
4	2	File size
8	3	File last modified date (in text)
16	4	File last modified time (in text)
32	5	File attributes (ADHRS)
64	6	File last modified date/time combination (in seconds since January 1, 1970)
128	7	File creation date (in text)
256	8	File creation time (in text)
512	9	File creation date/time combination (in seconds since January 1, 1970)

The attributes returned as a result of bit 5 being set are printed as the capital letters A (archive) D (subdirectory), H (hidden), R (read-only), and S (system).

Comments: This function returns an array of text values listed in reverse alphabetical order. Each text value contains the information specified by Option for each file that matches

both Path and Attributes. If no files are found, the return value will be set to invalid. Notice that the only difference between this function and the FileFind function is that FileFind searches down through the whole directory tree, while this function looks in the immediate directory only. If you are looking for a "browse for folder" dialog box, please refer to the FileDialogBox function.

Example:

```
If ! ValidValid(datFiles);
[
  datFiles = Dir("G:\Research\*.DAT" { Path },
                 0 { All normal files },
                 1 { Retrieve file name });
]
```

To display the first entry, try:

```
ZText(10, 10, datFiles[0], 15, 0);
```

The result might be something like:

```
"TEST.DAT"
```

DirectApply

Description: Applies a set of changes directly to the repository, without disturbing existing (non-conflicting) changes already on either branch.

Returns: Module

Usage: Script Only.

Function Groups: Configuration Management

Related to: CommitEditedFiles

Format: LayerModule\DirectApply(AlreadyHasLock, LocalChangeSet, RSema, pComment)

Parameters:

AlreadyHasLock

Required Boolean. Set TRUE when we already have the

working copy lock.

RSema

Required. Repository semaphore, if already held by the caller.

CallBack

Required. Module name of the caller performing the changes.

FileSet

The set of files to be changed, identified using a path relative to the working-copy.

User

The user ID that is to be applied to the change.

Comment

Any text value that is the comment to be stored in the version log for this change.

pFail

A pointer to a Boolean. This will be set TRUE on failure.

Comments: This function uses a callback system in order to acquire a set of changes against the repository tip versions of a file or group of files (both the local and deployed versions) and then applies those changes directly to the local and deployed repository tips without altering the working copy.

The changes are then applied to the working copy as a result of the repository update. The primary purpose of this operation is to allow changes to a file to be deployed without also deploying any existing local changes to the same file. The local version must also be updated such that the file is updated correctly when the current local and deployed tips

differ – otherwise conflicts could result when the repository attempts to rationalize the two.

DirectApply is used to distribute operation changes, notes, security information, and other data for which it is not appropriate that one machine may be configured differently than its peers; it should not be used for normal configuration changes. To make all configuration changes deploy immediately set the "Automatically Deploy" flag in the CM Information panel.

The callback definition follows:

```
Callback (
  DeployChangeSet { Set of deploy buffers, make
changes to these };
  LocalChangeSet { Set of local buffers, make
changes to these };
  RSema { Repo semaphore };
  pComment { Comment pointer allowing late edit-
ing };
 )
```

This module launches a worker module into the Layer so that the operation is not interrupted by this module's caller being slain. In the case that the current machine is not supporting an open local branch the LocalChangeSet callback parameter will be invalid.

Examples:

Disable

Note: Deprecated. Do not use in new code.

(Alarm Manager module)

Description: Use AlarmManager\EvaluateAlarm in all new code.

Told the Alarm Manager to disable an alarm. Disable will also clear any active or unacknowledged state that might exist.

Returns: 0

Usage: Script Only.

Function Groups: Alarm

Related to: Register (Alarm Manager) | CurrentTime

Format: \AlarmManager\Disable(AlarmObject[, EventTime]);

Parameters:

AlarmObject

Required. The object value for the alarm that was passed to the Register subroutine that will be disabled.

EventTime

Optional. The time stamp to use when adding this event to the alarm lists. If invalid or not defined, the default is CurrentTime().

Comments: The Disable subroutine always returns "0".

DisconnectFromMachine

(RPC Manager Library)

Description: This subroutine disconnects from a workstation by decrementing the usage count on the specified workstation and forcing the RPC Manager to attempt to establish a connection with the specified workstation if it is not already connected. Subroutine call only.

Returns: Nothing

Usage: Script Only.

Function Groups: Network

Related to: ConnectToMachine | GetServer | GetServersListed |

GetStatus | IsClient | IsPotentialServer | IsPrimaryServer | Register (RPC Manager) | Send| SetRemoteValue

Format: \RPCManager\DisconnectFromMachine(Workstation)

Parameters:

Workstation

Required. Any of the name or IPs that can be used to connect to the workstation.

Comments: This subroutine is a member of the RPC Manager's Library, and must therefore be prefaced by \RPCManager\, as shown in the "Format" section. If the application you are developing is a script application, the subroutine call must be prefaced by System\RPCManager\, and the System variable must be declared in AppRoot.src.
When the usage count of the workstation reaches "0", the RPC Manager will not attempt to re-establish a connection to the workstation once the current connection is lost.

It is critical that each DisconnectFromMachine call should be paired with a ConnectToMachine call. If the number of DisconnectFromMachine calls exceeds the number of ConnectToMachine calls, the RPC Manager will not behave as expected and connection with the remote workstation may be impeded. A negative value for the Srv value in the socket's entry in the RPC Diagnostics Window may be an indication of a DisconnectFromMachine/ConnectToMachine mismatch.

Example:

```
If Valid(sNode) Done;
[
  \RPCManager\DisconnectFromMachine(sNode);
]
```

This closes an existing socket by using its socket value.

.

DLL

Description: Returns a value of a type specified by its parameter from a call to Microsoft Windows™ dynamic link library using the C calling convention.

Warning: For use by advanced users only. Great care must be taken that all parameter values are correct when using this statement, since incorrect usage may cause a system crash.

Returns: varies (see table under ReturnType)

Usage: Script Only.

Function Groups: DLL

Related to: LoadDLL

Format: DLL(DLLName, FuncName, ReturnType, BuffLen, Type1, Val1, Type2, Val2, ...)

Parameters:

DLLName

Required. Any text expression that gives the full path, file name, and extension of the DLL to load, or the handle returned from a LoadDLL statement.

FuncName

Required. Any text expression that gives the name of the function to call in the DLL.

ReturnType

Required. The return type of the DLL function, as shown in the following table

ReturnType	Attribute
0	Void (return value only)
1	16 bit Integer
2	32 bit Integer
3	64 bit Double
4	Pointer
5	HWnd value of VTScada Object

BuffLen

Required. Any numeric expression that gives the length of the returned buffer, if ReturnType is 4.

Type1, Val1, Type2, Val2, ...

Required. Are any expressions that determine the type of parameter passed to the DLL function, as shown in the table for the ReturnType parameter. The corresponding ValN parameter will be cast as this type before it is passed to the DLL function.

Comments: This statement allows a wide variety of other routines and code to be used in an application.

Note: 64-bit VTScada can load only 64-bit DLLs. 32-bit VTScada can load only 32-bit DLLs.

Note: VTScada Internet Clients can load only 32-bit DLLs regardless of whether the VTScada server is 32-bit or 64-bit.

DoLoop

Description: Executes a do-while loop in a script.

Returns: Nothing

Usage: Script only.

May be used in optimized Tag Parameter Expressions.

Function Groups: Logic Control

Related to: Case | Cond | IfElse | IfThen | WhileLoop

Format: DoLoop(Function1, Function2, …, Condition)

Parameters:

Function1, Function2, …

Required. Any expression or statement. This is the body of the loop. All of the Function parameters are executed in order prior to testing the Condition parameter.

Condition

Required. Any logical expression for the loop control. As long as this is true, the loop will repeat.

Comments: The Function parameters are executed at least once, then Condition is checked. If Condition is true, the Function parameters are executed and Condition is executed again. This repeats until Condition is false.

Note: Great care must be taken with this statement since all other VTScada statements cannot execute until the loop is complete. This statement has the potential to lock up the system if the Condition never becomes false.

Note: While the DoLoop statement has its place in VTScada programming, it should be noted that speed can be enhanced by a factor of approximately 5 through the use of array processing functions (please refer to "Array Processing" for further details). Array functions are listed in "Array Functions".

Example:

```
i = 0;
If 1 Main;
[
  DoLoop(x[i] = Concat("Pump ", i),
                        i++,
```

```
                    i < 10);
]
```

This loop executes 10 times, assigning a text string to the next element of x on each iteration.

DragHandle

Description: Drags a graphic object's selected handle.

Returns: Nothing

Usage: Steady State only.

Function Groups: Graphics

Related to: DragState (Obsolete)

Format: DragHandle(Object, X, Y, KeepRatio)

Parameters:

Object

Required. Any expression that gives the object value that defines the selected graphic list to drag.

X

Required. Any numeric expression that is the new X coordinate of the selected handle(s).

Y

Required. Any numeric expression that is the new Y coordinate of the selected handle(s).

KeepRatio

Required. Any logical expression. If true (non-zero), the aspect ratio of the graphic will be preserved during stretching (online). If false (0), it will not.

DrawArcPath

Description: Draws an arc in any window.

Returns: Nothing

Usage: Steady State only.

Function Groups: Graphics

Related to: DrawChordPath | DrawEllipticalPath | DrawPath | DrawPiePath

Format: DrawArcPath(Left, Bottom, Right, Top, RotateAngle, Object, StartX, StartY, EndX, EndY)

Parameters:

Left

Required. Any numeric expression for the left side of the arc's bounding box.

Bottom

Required. Any numeric expression for the bottom side of the arc's bounding box.

Right

Required. Any numeric expression for the right side of the arc's bounding box.

Top

Required. Any numeric expression for the top side of the arc's bounding box.

RotateAngle

Required. Any numeric expression for the angle of rotation of the arc in degrees. This represents a rotation about the center of the bounding box. If it is greater than 0, the arc's shape will still be defined by the proportions of the bounding box, but will no longer be within its (un-rotated) screen coordinates. Note that although the arc itself will rotate RotateAngle degrees, neither of the endpoints for the lines defined by StartX, StartY, and EndX, EndY will be affected (i.e. the arc will not 'line up' with the angled lines).

Object

Required. Any expression for the object value that defines the window in which to draw the arc.

StartX

Required. Any numeric expression for the X coordinate of an endpoint of a line that defines the starting angle. The other endpoint is the center of the arc's bounding box.

StartY

Required. Any numeric expression for the Y coordinate of an endpoint of a line that defines the starting angle. The other endpoint is the center of the arc's bounding box.

EndX

Required. Any numeric expression for the X coordinate of an endpoint of a line that defines the ending angle. The other endpoint is the center of the arc's bounding box.

EndY

Required. Any numeric expression for the Y coordinate of an endpoint of a line that defines the ending angle. The other endpoint is the center of the arc's bounding box.

Comments: This statement is intended for building drawing tools. The arc is drawn in white and is exclusive OR'ed onto the screen.

Example:

```
DrawArcPath(0, 500, 700, 0 { Bounding box for the arc },
            0 { No rotation occurs },
            Self() { Draw in this module's window },
            XLoc(), YLoc() { One angled line follows mouse },
            750, 300 { Other line of angle });
```

This draws an arc on the screen that has a horizontal line at its 0 degree position and whose second line of its open angle follows the mouse. As

the mouse moves and changes the angle of the two lines, the arc will follow its path, and more or less of it will appear.

DrawChordPath

Description: Draws a chord in any window.

Returns: Nothing

Usage: Steady State only.

Function Groups: Graphics

Related to: DrawArcPath | DrawEllipticalPath | DrawPath | DrawPiePath

Format: DrawChordPath(Left, Bottom, Right, Top, RotateAngle, Object, StartX, StartY, EndX, EndY)

Parameters:

Left

Required. Any numeric expression for the left side of the bounding box.

Bottom

Required. Any numeric expression for the bottom side of the bounding box.

Right

Required. Any numeric expression for the right side of the bounding box.

Top

Required. Any numeric expression for the top side of the bounding box.

RotateAngle

Required. Any numeric expression for the angle of rotation of the chord in degrees. This represents a rotation about the center of the bounding box. If it is greater than 0, the chord's shape will still be defined by the pro-

portions of the bounding box, but will no longer be within its (un-rotated) screen coordinates. Note that although the chord itself will rotate RotateAngle degrees, neither of the endpoints for the lines defined by StartX, StartY, and EndX, EndY will be affected (i.e. the chord will not 'line up' with the angled lines).

Object

Required. Any expression for the object value that defines the window in which to draw the chord.

StartX

Required. Any numeric expression for the X coordinate of an endpoint of a line that defines the starting angle. The other endpoint is the center of the bounding box.

StartY

Required. Any numeric expression for the Y coordinate of an endpoint of a line that defines the starting angle. The other endpoint is the center of the bounding box.

EndX

Required. Any numeric expression for the X coordinate of an endpoint of a line that defines the ending angle. The other endpoint is the center of the bounding box.

EndY

Required. Any numeric expression for the Y coordinate of an endpoint of a line that defines the ending angle. The other endpoint is the center of the bounding box.

Comments: This statement is intended for building drawing tools. The chord shape is drawn in white and is exclusive OR'ed onto the screen. The shape of the object is such that the angle defined by the two lines forms a flattened side to an ellipse (if the angle between the lines is less than 180 degrees), or a sliver that has been cut from the ellipse (if the angle is greater than 180 degrees).

Example:

```
DrawChordPath(0, 500, 700, 0 { Bounding box for the chord },
              90 { Rotate 90 degrees clockwise },
              Self()  { Draw in this module's window },
              550, 470 { One line of angle },
              750, 300 { Other line of angle });
```

This draws an small chord in the upper right hand corner of the screen, with its defining angle 90 degrees out of phase in the lower right hand corner.

DrawEllipticalPath

Description: Draw an ellipse in any window.

Returns: Nothing

Usage: Steady State only.

Function Groups: Graphics

Related to: DrawArcPath | DrawChordPath | DrawPath | DrawPiePath

Purpose: This statement draws an ellipse in any window.

Format: DrawEllipticalPath(Left, Bottom, Right, Top, RotateAngle, Object)

Parameters:

Left

Required. Any numeric expression for the left side of the bounding box.

Bottom

Required. Any numeric expression for the bottom side of the bounding box.

Right

Required. Any numeric expression for the right side of the bounding box.

Top

Required. Any numeric expression for the top side of

the bounding box.

RotateAngle

Required. Any numeric expression for the angle of rotation of the ellipse in degrees. This represents a rotation about the center of the bounding box. If it is greater than 0, the ellipse's shape will still be defined by the proportions of the bounding box, but will no longer be within its (un-rotated) screen coordinates.

Object

Required. Any expression for the object value that defines the window in which to draw the ellipse.

Comments: This statement is intended for building drawing tools. The ellipse is drawn in white and is exclusive OR'ed onto the screen.

Example:

```
DrawEllipticalPath(200, 300, 640, 190
              { Bounding box for the ellipse },
              65 { Rotate 65 degrees clockwise },
              Self() { Draw in this module's window });
```

This draws a stubby, cigar-shaped ellipse in the center of the screen. It is situated in a NE-SW orientation.

DrawPath

Description: Draws a polygon in any window.

Returns: Nothing

Usage: Steady State only.

Function Groups: Graphics

Related to: DrawArcPath | DrawChordPath | DrawEllipticalPath | DrawPiePath | PathDraw

Format: DrawPath(XArray, YArray, N, Object)

Parameters:

XArray

Required. An array element expression. This array specifies the X coordinates of the path to draw, beginning with the element specified.

YArray

Required. An array element expression. This array specifies the Y coordinates of the path to draw, beginning with the element specified.

N

Required. Any numeric expression for the number of points (number of array elements) in the path.

Object

Required. Any expression for the object value that defines the window.

Comments: This statement is intended for building drawing tools. The polygon is drawn in white and is exclusive OR'ed onto the screen.
Do not confuse DrawPath with the function, PathDraw()

Example:

```
DrawPath(xValues[0], yValues[0]
        { Starting coordinates that define polygon },
        11 { Figure has 11 vertices },
        Self() { Draw in this module's window });
```

This will draw a 10 sided polygon as defined by the coordinates in the two arrays.

DrawPiePath

Description: Draws a pie in any window.

Returns: Nothing

Usage: Steady State only.

Function Groups: Graphics

Related to: DrawArcPath | DrawChordPath | DrawPath | DrawPiePath

Format: DrawPiePath(Left, Bottom, Right, Top, RotateAngle, Object, StartX, StartY, EndX, EndY)

Parameters:

Left

Required. Any numeric expression for the left side of the bounding box.

Bottom

Required. Any numeric expression for the bottom side of the bounding box.

Right

Required. Any numeric expression for the right side of the bounding box.

Top

Required. Any numeric expression for the top side of the bounding box.

RotateAngle

Required. Any numeric expression for the angle of rotation of the pie, in degrees. This represents a rotation about the center of the bounding box. If it is greater than 0, the pie's shape will still be defined by the proportions of the bounding box, but will no longer be within its (un-rotated) screen coordinates. Note that although the pie itself will rotate RotateAngle degrees, neither of the endpoints for the lines defined by StartX, StartY and EndX, EndY will be affected (i.e. the pie will not 'line up' with the angled lines).

Object

Required. Any expression for the object value that defines the window.

StartX

Required. Any numeric expression for the X coordinate of an endpoint of a line that defines the starting angle. The other endpoint is the center of the bounding box.

StartY

Required. Any numeric expression for the Y coordinate of an endpoint of a line that defines the starting angle. The other endpoint is the center of the bounding box.

EndX

Required. Any numeric expression for the X coordinate of an endpoint of a line that defines the ending angle. The other endpoint is the center of the bounding box.

EndY

Required. Any numeric expression for the Y coordinate of an endpoint of a line that defines the ending angle. The other endpoint is the center of the bounding box.

Comments: This statement is intended for building drawing tools. The pie is drawn in white and is exclusive OR'ed onto the screen.

Example:

```
DrawPiePath(10, 325, 430, 75
           { Bounding box for the pie },
           90 { Rotate 90 deg. counter-clockwise },
           Self() { Draw in this module's window },
           550, 470 { One line of angle },
           750, 300 { Other line of angle });
```

This draws a wedge of pie on the left side of the screen, point upward.

DrawScale

(Meter Parts Library)

Description: Will draw a scale (i.e. tick marks) for a linear or radial type meter. These marks are images (normally lines) indicating the major and minor divisions of the entire scale. This func-

tion must be called inside a GUITransform in order to work properly.

Returns: Nothing

Usage: Steady State only.

Function Groups: Graphics

Related to:

Format: \MeterParts\DrawScale(MajorTickImage, MinorTickImage, MajorDivisions, MinorDivisions, RelativeSize, LinearScale, Orientation, MinAngle, MaxAngle, Hue, Saturation, Brightness, Transparency, Contrast, ColorizeHue, ColorizeIntensity)

Parameters:

MajorTickImage

Required. The full path to an image file to use as a major tick mark.

MinorTickImage

Required. The full path to an image file to use as a minor tick mark.

MajorDivisions

Optional. The number of major divisions in the scale. The default is 1 major division.

MinorDivisions

Optional. The number of Minor divisions between the major divisions in the scale. The default is 5 minor divisions.

RelativeSize

Optional. The relative size of the tick marks. If set, the tick marks will resize independently from the automatic sizing that is done when you resize the transform. Typically the range for this parameter is from 0 to 2, where 2 is double the default size and 0.5 is half

the default size. The default is 1 which is the native size as scaled with the transform size.

LinearScale

Optional. A flag that, when set to true causes a linear scale to be drawn. When set to false, a Radial scale will be created. The default is true.

Orientation

Optional. A flag that is relevant only when drawing a linear type scale. When set to true the scale will be oriented vertically. When set to false, the scale will be oriented horizontally. The default is false.

MinAngle

Optional. The angle for the starting position of a radial type scale. 0 is defined as up or the 12 o'clock position. This parameter is relevant only when drawing a radial type scale. The default is 0.

MaxAngle

Optional. The angle for the ending position of a Radial type scale. . 0 is defined as up or the 12 o'clock position. This parameter is relevant only when drawing a radial type scale. The default is 90.

Hue

Optional. The Hue translation to perform on the tick mark image. The image must have color in it already in order to perform a hue translation. If there is no color to start with, then changing this value does nothing. You can add color by setting a value for the ColorizeHue parameter, described later.
The default is 0, indicating that no hue translation is done and the indicator is in its native color.

Saturation

Optional. The amount of saturation of the colors in the tick mark image. A value of 0 will make the image

black and white (no color saturation). A value of 2 produces a brightly colored (saturated) indicator. The default is 1 which corresponds to the native saturation of the indicator image.

Brightness

Optional. An adjustment of the brightness of the tick mark image. Higher numbers produce a brighter image. A 0 produces a black image. The default is 1 which corresponds to the native brightness of the tick mark image.

Transparency

Optional. An adjustment of the opacity of the tick mark image where 1 means 100% opacity and 0 means %100 transparent. The default is 1.

Contrast

Optional. An adjustment of the contrast of the colors in the tick mark image. A value of 0 produces a flat looking image and a value of 2 gives a high contrast image. The default is 1 which corresponds to the native contrast of the image.

ColorizeHue

Optional. A value that works in conjunction with ColorizeIntensity. This is the hue of the color that is introduced by colorizing an image. Colorizing an image will introduce color into an image that previously was black and white or grayscale. The default value is 0.

ColorizeIntensity

Optional. A value to define how much color to introduce into the image. The default is 0, meaning not to introduce any color at all into the image.

Comments: This function must be called from within a GUITransform statement in order for it to work correctly.
The Tick Marks are scaled according to the size of the

transform and the RelativeSize parameter.

Example:

```
 GUITransform(674, 284, 824, 134,
              1, 1, 1, 1, 1 { Scaling             },
              0, 0          { Movement            },
              1, 0          { Visibility, Reserved },
              0, 0, 0       { Selectability       },
              Variable("Code\MeterParts")\DrawScale("Bitmaps\Meter
Parts\Tick Marks\Large_Thin.png", "Bitmaps\Meter Parts\Tick Mark-
s\Small_Thin.png", 1, 5, 1, 1, 0, 0, 90, 0, 1, 1, 1, 1, 0, 0));
```

DriveInfo

Description: Returns information about a disk drive.

Returns: Pointer (see comments)

Usage: Script Only.

Function Groups: Software and Hardware

Related to: Dir | FileFind

Format: DriveInfo(Attributes, Option)

Parameters:

Attributes

Required. Any numeric expression which gives the attributes to match for each drive. Drives that don't match these attributes won't be listed. The Attributes parameter is constructed by adding together the num-

Attribute	Bit No.	Drive Type
1	0	Removable drive (floppy disk)
2	1	Fixed drive (hard disk)
4	2	Remote drive (network disk)
8	3	CD-ROM drive

Option

Required. Any numeric expression giving the type of text information to generate. All information is placed in a single text string (which will be stored in an element of the array that is created). The information is written from left to right, with lowest option numbers first.

Option	Bit No.	Information Type
1	0	Drive type
2	1	Drive letter and colon (e.g. C:)
4	2	Volume name
8	3	Total volume space (in bytes)
16	4	Volume space available (in bytes) for consumption

Comments:

This function will only return information on network drives that have been assigned drive letters (i.e. those having been used in a Windows™ command prompt "net use" statement or VTScada Redirect statement).

This function returns an array of text values (one element per drive). Each text value contains the information specified by Option for each drive that matches Attributes. The size information is expressed as a text string of a decimal number of the requested value concatenated with the other

Value	Drive Type
0	Removable drive (floppy disk)
1	Fixed drive (hard disk)
2	Remote drive (network disk)
3	CD-ROM drive

Example:

```
If ! Valid(drives);
[
  drives = DriveInfo(6 { All hard and network disks },
  3 { Drive type and letter });
]
Table(drives[0] { Start at beginning of array },
      ArraySize(drives, 0){ Number of elements in array },
      10, 10, 0, 10 { First line point, vertical list },
      4, 0, 100 { Text values; min/max chars },
      15, 0, 0 { White on black background },
      0, 0 { Standard height, no rotation });
```

This would list the drive types and letters, both fixed and remote, available on the system in table form in the upper left hand portion of the screen.

Droplist

(System Library)

Description: Draws a droplist with (optional) title or bevel or both.

Returns: Nothing

Usage: Steady State only.

Function Groups: Graphics

Related to: Bevel | CheckBox | ColorSelect | DropTree | GridList | HScrollbar | Listbox | RadioButtons | Spinbox | SplitList | VScrollbar

Format: \System\DropList(X1, Y1, X2, Y2, Data, Title, Index, FocusID, Trigger, NoEdit, Init, Variable [, DrawBevel, VertAlign, AlignTitle, Style, BGColor, FGColor])

Parameters:

X1

Required. Any numeric expression giving the X coordinate on the screen of one side of the droplist.

Y1

Required. Any numeric expression giving the Y coordinate on the screen of either the top or bottom of the (opened) droplist.

X2

Required. Any numeric expression giving the X coordinate on the screen of the side of the droplist opposite to X1.

Y2

Required. Any numeric expression giving the Y coordinate on the screen of the top or bottom of the (opened) droplist, whichever is the opposite to Y1.

Data

Required. An array of data to be displayed in the droplist.

Title

Required. Any text expression to be used as a title for the droplist.

Index

Required. A variable whose value indicates the array index of the highlighted item in the list. Index can be set invalid to reset the droplist.

FocusID

Required. Any numeric expression for the focus number of this graphic. If this value is 0, the droplist will display its current setting, but will not be able to be opened (i.e. its value cannot be changed) and will appear grayed out. The default value is 1.
If this parameter is invalid, keyboard input (such as the carriage return key) will be ignored.

Trigger

Required. If the droplist is editable, Trigger provides feedback. While editing, the value will be 0. When edit-

ing is complete (tab, enter or loss of focus) the value will change to non-zero; 1 if enter is pressed, 2 otherwise.
If this information is not required and the next parameter is used, a value of invalid or a constant may be substituted.

NoEdit

Required. Any logical expression. If TRUE (non-0) the text displayed in the droplist cannot be edited directly, if FALSE (0) it can be edited in the same manner as an editfield.

Note: if the provided variable is declared but left INVALID (i.e. a C++ BASEVALUE class) then NoEdit will default to TRUE.
If the NoEdit parameter is set directly to INVALID (a C++ VALUE class) then NoEdit will default to FALSE.

Init

Required. Any expression for the initial value displayed in the field if Index is set to invalid. If NoEdit is true, then Init must be an element of the data array.

Variable

Required. The variable whose value is set by the droplist.

DrawBevel

An optional parameter that is any logical expression. If true (non-0) a bevel is drawn around the droplist, if false (0) no bevel is drawn. The default value is false.

VertAlign

An optional parameter that is any numeric expression that sets the vertical alignment of the editfield according to one of the following options

VertAlign	Vertical Alignment
0	Top
1	Center
2	Bottom

Whether or not the title is included when the vertical alignment is calculated is determined by the value of AlignTitle. The default value is 0.

AlignTitle

An optional parameter that is any logical expression. If true (non-0) the title is included in the calculation for vertical alignment, if false(0) it is added to the droplist after it (and its bevel if one exists) has been vertically aligned. The default is true.

Style

Comprised of a combination of bit values to yield the desired effects.
Bits 0 and 1 define mutually exclusive styles of operation. They can be set to one of the following values:

Bit Number	Definition
0	No droplist, rather a listbox with the selected item above
1	Droplist and editable selection
2	Droplist with non-editable selection

Bits 2 and 3 define input character handling. If neither is set, input is passed to script code as typed.

Bit Number	Definition
2	Input is converted to all uppercase.
3	Input is converted to all lowercase.

Bit 4 controls list sorting.

Bit Number	Definition
4	The list is presented in sorted order to the user.

Bit 8 enables Windows visual styles.
Bit 9 defines and enables exact use of application-defined geometry. The default is to perform auto-geometry modification to give the best fit.

Bit Number	Definition
9	Enable application-defined geometry.

BGColor

Optional. Any numeric expression for the background color of the control. No default value.

FGColor

Optional. Any numeric expression for the foreground color of the control. No default value.

Comments:

This module is a member of the System Library, and must therefore be prefaced by \System\, as shown in the "Format" section. If you are developing a script application, use "System\..." rather than "\System\..." in the function call.

The height of the (unopened) droplist is constant, with X1 and X2 defining its width, and Y1 and Y2 defining its opened height, which may or may not include the added height of a title and bevel, depending on the alignment used and if they exist. Note that if the entire list can be displayed in a smaller area than indicated by Y1 and Y2, the dropped list height will be decreased accordingly. Droplist enforces a minimum dropped list size of four rows, unless the list is smaller in size.

Droplist may not be wrapped in a GUITransform.

Example:

```
Init [
If 1 Main;
  [
    Choices = New(4);
    Index = 0;
    Choices[0] = "Open";
    Choices[1] = "Close";
    Choices[2] = "On";
    Choices[3] = "Off";
  ]
]
```

```
Main [
  System\DropList(10, 210, 210, 390 { Boundaries of list },
                  Choices { Data displayed },
                  "Match String" { Title },
                  Index { Highlighted index },
                  1, 1 { Focus ID, trigger },
                  0 { Editable field },
                  "Cancel" { Starting value },
                  Val { Variable to set },
                  0 { No bevel },
                  0 { Align top of list });
]
```

This shows an editable droplist without a bevel, with the top of the droplist itself at Y = 210; the title "Pick a String" is above it (i.e. beyond the top of the defined area). The initial value in the field, and thus the initial value of Val will be "Cancel"; when the list is first opened, the first array entry (element 0) will be highlighted. The trigger variable is not used.

DropTree

(System Library)

Description: Draws a control similar to a Droplist, but rather than a flat list, a tree of possible selections is displayed.

Returns: Object reference.

Usage: Script Only.

Function Groups: System

Related to: Droplist |

Format: \System\DropTree(Left, Bottom, Right, Top, TreeData, Title, SelectedKey, FocusID, FTrigger[, DrawBevel, AlignTitle, DlgRoot, Trigger])

Parameters:

Left

Any numeric expression for the left coordinate of the tool.

Bottom

Any numeric expression for the bottom coordinate of the tool.

Right

Any numeric expression for the right coordinate of the tool.

Top

Any numeric expression for the top coordinate of the tool.

TreeData

The data to display, in Node array format. The format is the same as for a call to TreeControl.

Title

Any text expression to be used as a title.

SelectedKey

The key of the selected item. Must be a variable, and may specify initial selection.

FocusID

Any numeric expression for the FocusID.

FTrigger

The Focus Trigger of the DropTree.

DrawBevel

Optional Boolean. If TRUE (non-0) a bevel is drawn. Defaults to FALSE.

AlignTitle

An optional parameter that is any logical expression. If TRUE (non-0) the title is included in the calculation for vertical alignment. The default is FALSE.

DlgRoot

Optional. The object value of the root dialog. Used for alignment of the DropTree.
Defaults to the caller if not specified.

Trigger

An optional numeric expression. Initially set to zero (0) when the DropTree opens.
If the user presses the Escape key or closes the extended window, then Trigger becomes 1.

Comments: This function allows the use of disabled options - grayed in appearance and unselectable.
The following set of helper functions is available. These may be added to the caller of DropTree in order to add special handling of certain events.

- OnLeftClick(NodeArray)
  - Subroutine: The left mouse button was released over a tree node.
- OnRightClick(NodeArray, X, Y)
  - Subroutine: The right mouse button was released over a node. X and Y are the coordinates of the mouse.
- OnDoubleClick(NodeArray, X, Y)
  - Subroutine: The left mouse button was double-clicked over a node. X and Y are the coordinates of the mouse.
- CreateSubTree(NodeKey)
  -
- ExpandTreeToNode(NodeKey)
  - Recursive Subroutine: Used for SetSelected call, expands to given node
- Collapse()
  - Will traverse the whole tree starting from the leaves, working towards the root, calling CollapseNodes() on each node.

Related Information:

TreeControl Module in the VTScada Programmer's Guide.

E Functions

The sections that follow identify all VTScada functions beginning with "E".

Edge

Description: Test for a rising or falling edge.

Returns: Boolean

Usage: Steady State only.

Function Groups: Generic Math, Variable

Related to: Change

Format: Edge(Value, Mode)

Parameters:

Value

Required. Any expression giving a numeric status value to be tested. It is interpreted as true if it is non-zero, and false if it is zero.

Mode

Required. Dictates whether the change from false to true or true to false is tested. If it is 0, a true to false (a falling edge) is tested. If it is 1, a false to true (a rising edge) is tested.

Comments: Once triggered, the return value will remain true until the function is reset. Typically, Edge would be used as an expression in a function such as Latch that will reset its parameters.

Example 1:

```
If Edge(Var1 > 2, 1) NextState;
```

Example 2:

```
Latch(Edge(Variable("Tag_1")\Value > 10,1),
TimeOut(Watch(0, Value), 2)) ?  1 : 0
```

When the value of the tag, ‘Tag_1’ changes from less than 10 to greater than 10, the value of the expression will change to 1.
The change in the expression value will start the Timeout function, which will reset the expression after 2 seconds and allow the process to repeat the next time Tag_1’s value increases past 10.

Related Information:

See: "Latching and Resetting Functions" in the VTScada Programmer's Guide

Edit

(System Library)

Description: Draws an edit field with (optional) title or bevel or both.

Returns: Nothing

Usage: Steady State only.

Function Groups: Graphics

Related to: CheckBox | ColorSelect | Droplist | Listbox | RadioButtons | Spinbox

Format: \System\Edit(X1, Y1, X2, Y2, Title, Variable [, FocusID, Trigger, View, DataType, DrawBevel, VertAlign, AlignTitle, LowLimit, HighLimit, Style, PrefixValue, SuffixValue, BGCo-lor, FGColor])

Parameters:

X1

Required. Any numeric expression giving the X coordinate on the screen of one side of the edit field.

Y1

Required. Any numeric expression giving the Y coordinate on the screen of either the top or bottom of the edit field.

X2

Required. Any numeric expression giving the X

coordinate on the screen of the side of the edit field opposite to X1.

Y2

Required. Any numeric expression giving the Y coordinate on the screen of the top or bottom of the edit field, whichever is the opposite of Y1.

Title

Required. Any text expression to be used as a title for the field.

Variable

Required. A variable whose value is set by the edit field.

FocusID

Optional. Any numeric expression for the focus number of this graphic. If this value is 0, the edit field will display its current setting, but its value will not be able to be changed and it will appear grayed out. The default value is 1.

Trigger

Optional. Trigger provides feedback. While editing, the value will be 0. When editing is complete (tab, enter or loss of focus) the value will change to non-zero; 1 if enter is pressed, 2 otherwise.

View

Optional. Indicates how to display the edit field, as follows

View	Display Mode
0	Invisible
1	Normal (color scheme - no graying)
2	Grayed-out (only if FocusID is 0)

This parameter may be used to force an edit

field with a FocusID of 0 to be displayed normally, rather than allowing it to default to its grayed color. Note that if the FocusID is not 0, setting this value as 2 will not force the field to gray out.
The default value is 2 if FocusID is 0 and 1 otherwise.

DataType

Optional. Any numeric expression giving the type of data accepted by the edit field as follows

DataType	Description
0	Byte (unsigned)
1	Short (2 byte signed)
2	Long (4 byte signed)
3	Double precision floating point (8 byte signed)
4	Text
5	Octal (4 byte unsigned)
6	Hexadecimal (4 byte unsigned)

For types 0 - 2, if the number entered into the field is prefaced by a "0x" the value is taken to be hexadecimal format, and if it is prefaced by a "0" it is considered to be octal. In either case, the value is converted to decimal format when return is pressed or the focus is lost.
For type 5, regardless of whether or not the number entered into the field is prefaced by a "0" the value is taken to be octal and will be displayed as such. The actual type of Variable will be text.

Type 6, like type 5 will be kept in its declared format of hexadecimal regardless of whether or not the number entered into the field is prefaced by a "0x". The actual type of Variable will be text

The default value is 4 - text.

DrawBevel

Optional. Any logical expression. If true (non-0) a bevel is drawn around the edit field, if false (0) no bevel is drawn.

If the edit field is beveled, its size will become fixed and will be the same as that for a droplist. The default value is false.

VertAlign

Optional. Any numeric expression that sets the vertical alignment of the edit field according to one of the following options

VertAlign	Vertical Alignment
0	Top
1	Center
2	Bottom

Whether or not the title is included when the vertical alignment is calculated is determined by the value of AlignTitle. The default value is 0.

AlignTitle

Optional. Any logical expression. If true (non-0) the title is included in the calculation for vertical alignment. If false(0) it is added to the edit field after it (and its bevel if one exists) has been vertically aligned. The default is true.

LowLimit

Optional. Any expression giving the minimum value or minimum number of characters to be accepted by the edit field (depending on the data type).
This value may be a decimal, octal or hexadecimal value. If this parameter is valid and a value less than LowLimit is entered in the field (or there are too few characters, in the case of text value), the variable set by the field will revert to the previous value. No default

HighLimit

Optional. Any numeric expression giving the maximum value or maximum number of characters to be accepted by the edit field (depending on the data type).

- If used for numbers, the default is 255 characters.
- If used for text, the default is 32767 characters.

This value may be a decimal, octal or hexadecimal value. If this parameter is valid and a value greater than HighLimit is entered in the field (or there are too many characters, in the case of text value), the variable set by the field will revert to the previous value. No default

Style

Optional. Comprised of a combination of bit values to yield the desired effects. Default 0.

Bits 0 and 1 are reserved for bit compatibility with WinComboCtrl, and should be set to "0".

Bits 2, 3 and 4 define input character handling. If not set, input is exactly as typed.

Bit Number	Definition
2	Input is converted to all uppercase.
3	Input is converted to all lowercase.
4	Input is masked. Any characters typed will appear as asterisks. (useful for password fields)

Bit 5 controls multi-line edit controls.

Bit Number	Definition
5	Multi-line editing. When set, this bit causes a typed Enter key to be interpreted as "move to the start of the next line". Text that contains carriage-return & line-feed characters has a line break inserted at each set.

Bits 6 and 8 are reserved.

Bit 2^9 Not used. Height is fixed by constant, \EditHt or \TEditHt (with title).

PrefixValue

Optional. Indicates the text expression that should be displayed immediately before (i.e. to the left of) the editable part of the control. No default.

SuffixValue

Optional. Indicates the text expression that should be displayed immediately after (i.e. to the right of) the editable part of the control. No default.

BGColor

Optional. Any numeric expression for the background color of the control. No default value.

FGColor

Optional. Any numeric expression for the foreground color of the control. No default value.

Comments:

This module is a member of the System Library, and must therefore be prefaced by \System\, as shown in the "Format" section. If you are developing a script application, use "System\..." rather than "\System\..." in the function call.

The height of the edit field is constant, with X1 and X2 defining its width, and Y1 and Y2 defining the boundaries in which it is to be confined vertically, which may or may not include the added height of a title and bevel, depending on the alignment used and if they exist.

For any optional parameter that is to be set, all optional parameters preceding the desired one must be present, although they may be invalid.

Examples:

```
System\Edit(10, 50 { Top left corner - fixed },
            110, 20 { Bottom right corner },
            "Description" { Title },
            Desc { Var to update }
            { Remainder of parameters not used, so omitted });
```

```
System\Edit(20, 110, 120, 70 { Outline of field },
            "Name", Name { Title; var to update },
            2, DoneFlag { Focus ID, trigger },
            Invalid { Default display },
            2 { Long integer values },
            Invalid { Default of no bevel },
            1 { Center the editfield },
            1 { Include title in alignment },
            Invalid { No minimum value },
            100 { Maximum acceptable value });
```

EditFile

Description: Informs the configuration management system that a file has been modified in the working copy, typically before making a call to CommitEditedFiles.

Returns: Nothing

Usage: Script Only.

Function Groups: Configuration Management

Related to: CommitEditedFiles

Format: \LayerModule\EditFile(ModItem[, JustInformSubscribers])

Parameters:

ModItem

Required. A file name, a dictionary of file names, or an array of names of modified files.
File paths relative to the working copy are recommended. Files outside the working copy cannot be added.

JustInformSubscribers

Optional Boolean. Set TRUE to *not* mark the file as modified. This is useful when switching between repository revisions.Defaults to FALSE.

Comments: This function must be called only if the caller has the semaphore (working copy lock). The files will be added to the Modified dictionary so that it is distributed to all subscribers when the transaction is closed.
If a set of files is passed, then the values will also be passed to the repository. The primary purpose of this is that it allows for files to be marked as ignored.

Examples:

EditINI

(VTS Library)

Description: Draws an edit field from which a value of an application property in Settings.Startup may be set.

Returns: Nothing

Usage: Steady State only.

Function Groups: Graphics

Related to: EditINICheckBox | ReadINI | ReadSectINI | WriteSectINI

Format: \Library\EditINI(Section, VarName [, Title, DataType, FocusID, View, DrawBevel, VertAlign, AlignTitle, UpdateVar, LowLimit, HighLimit])

Parameters:

Section

Required. Any text expression giving the name of the section in the file. This should not include the square brackets delimiting the section.

VarName

Required. Any text expression giving the name of the variable for which the value is to be set.

Title

Optional. Any text expression to be used as a title for the field. No default.

DataType

Optional. Any numeric expression giving the type of data accepted by the edit field as follows. No default.

DataType	Type
0	Byte (unsigned)
1	Short (2 byte signed)
2	Long (4 byte signed)
3	Double precision floating point (8 byte signed)
4	Text

Note that for types 1 and 2, if the number entered into the field is prefaced by a "0x", the value is taken to be a hexadecimal value and is converted to a decimal value when return is pressed. The default value is 4.

FocusID

Optional. Any numeric expression for the focus number of this graphic. If this value is 0, the edit field will display its current setting, but its value will not be able to be changed and it will appear grayed out. The default value is 1.

View

Optional. Indicates how to display the edit field, as follows:

View	Display
0	Invisible
1	Normal (color scheme - no graying)
2	Grayed out (only if FocusID is 0)

This parameter may be used to force an editfield with a FocusID of 0 to be displayed normally, rather than allowing it to default to its grayed

color. Note that if the FocusID is not 0, setting this value as 2 will not force the field to gray out.

The default value is 2 if FocusID is 0 and 1 otherwise.

DrawBevel

Optional. Any logical expression. If true (non-0) a bevel is drawn around the editfield, if false (0) no bevel is drawn. If the editfield is beveled, its size will become fixed and will be the same as that for a droplist. The default value is true.

VertAlign

Optional. Any numeric expression that sets the vertical alignment of the editfield according to one of the following options:

VertAlign	Vertical Alignment
0	Top
1	Center
2	Bottom

Whether or not the title is included when the vertical alignment is calculated is determined by the value of AlignTitle. The default value is 0.

AlignTitle

Optional. Any logical expression. If true (non-0) the title is included in the calculation for vertical alignment. If false (0) it is added to the editfield after it (and its bevel if one exists) has been vertically aligned. The default is true.

UpdateVar

Obsolete. The running system is always updated.

LowLimit

Optional. Any expression giving the minimum value or minimum number of characters to be accepted by the editfield (depending on the data type).
This value may be a decimal, octal or hexadecimal value. If this parameter is valid and a value less than LowLimit is entered in the field (or there are too few characters, in the case of text value), the variable set by the field will revert to the previous value. no default:

HighLimit

Optional. Any numeric expression giving the maximum value or maximum number of characters to be accepted by the editfield (depending on the data type). This value may be a decimal, octal or hexadecimal value. If this parameter is valid and a value greater than HighLimit is entered in the field (or there are too many characters, in the case of text value), the variable set by the field will revert to the previous value.

Comments:

This module is a member of the VTS Library and must therefore be called from within a GUITransform and prefaced by \Library\.
The height of the edit field is constant, with the horizontal boundaries of its calling transform defining its width, and the vertical boundaries of its calling transform defining its height (to a minimum size). The height will include the height of the bevel, but may or may not include the title, depending on the alignment used.
For any optional parameter that is to be set, all optional parameters preceding the desired one must be present, although they may be invalid.

Example:

```
GUITransform(180, 172, 280, 142,
             1, 1, 1, 1, 1 { Scaling             },
             0, 0          { Movement            },
             1, 0          { Visibility, Reserved },
             0, 0, 0       { Selectability       },
             Variable("Code\Library")\EditIni("System",
```

```
GiveUpCallTimeout",
                                                    "Give up", 2, 1, 2, 1,
1, 1,
                                                    Invalid, 0, 2000));
```

This causes an edit box to be displayed with the title "Give up". The operator may use it to change the value of GiveUpCallTimeout between the range of 0 and 2000.

EditINICheckBox

(VTS Library)

Description: Draws an edit field check box, with which a value of an application property in Settings.Startup may toggled between true and false.

Returns: Nothing

Usage: Steady State only.

Function Groups: Graphics

Related to: EditINI | ReadINI | ReadSectINI | WriteSectINI

Format: \Library\EditINIcheck box(Section, VarName [, Title, BoxOnLeft, Align, FocusID, UpdateVar, EnableIfEditing])

Parameters:

Section

Required. Any text expression giving the name of the section in the file. This should not include the square brackets delimiting the section.

VarName

Required. Any text expression giving the name of the variable for which the value is to be set.

Title

Optional. Any text expression to be used as a title for the field. No default:

BoxOnLeft

Optional. A Boolean, indicating whether the check box

should be to the left or right of the label. Defaults to TRUE (box to the left of the label).

Align

Optional. Any numeric expression controlling the alignment of the label and the box. Defaults to 3 - Left, vertically centered.

Value	Meaning
0	Left, top.
1	Right, top.
2	Full, top.
3	Left, vertically centered.
4	Right, vertically centered.
5	Full, vertically centered.
6	Left, bottom.
7	Right, bottom.
8	Full, bottom.

FocusID

Optional. Any numeric expression for the focus number of this graphic. If this value is 0, the check box will display its current setting, but its value will not be able to be changed and it will appear grayed out. The default value is 1.

UpdateVar

Obsolete. The running system is always updated.

EnableIfEditing

Obsolete. Will always evaluate to FALSE. Within the Idea Studio, EditINICheckbox will always be disabled. Everywhere else, it will be enabled.

Comments: This module is a member of the VTS Library and must therefore be called from within a GUITransform and prefaced by \Library\.

Example:

```
GUITransform(495, 375, 595, 345,
               1, 1, 1, 1, 1 { Scaling              },
               0, 0          { Movement             },
               1, 0          { Visibility, Reserved },
               0, 0, 0       { Selectability        },
             \Library\EditIniCheckBox("System", "AlarmPopupsEnable",
"Alarm Popups", TRUE, 3, 1, Invalid, FALSE));
```

This draws a check box that operators may use to enable or disable the Alarm Popups application property.

Editor

Note: Deprecated. Do not use in new code.

Description Displays an editor on the screen.

Returns Nothing

Usage Steady State only.

Function Groups Editor, Graphics

Related to: AddEditorText | CurrentLine | ForceEvent | GoToOffset | MakeEditor | SetEditMode

Format Editor(Left, Bottom, Right, Top, EditorValue, FocusID, Font [, Info])

Parameters

Left

Required. Any numeric expression that defines the left hand side of the square area where the editor will operate.

Bottom

Required. Any numeric expression that defines the bottom of the square area where the editor will operate.

Right

Required. Any numeric expression that defines the right hand side of the square area where the editor will operate.

Top

Required. Any numeric expression that defines the top of the square area where the editor will operate.

EditorValue

Required. An editor value that has been created by MakeEditor which contains the text contents for the editor.

FocusID

Required. Any numeric expression giving the focus number of the graphic. If FocusID is zero, this graphic cannot receive the input focus.

Font

Required. Any expression that returns a (fixed pitch) font value. If the font supplied is not fixed pitch, the system font is used.

Info

Optional. A one-dimensional array where information on the editor will be recorded, as follows: (no default)

Info	Information
0	Current line number
1	Current column number
2	Total number of lines in editor
3	Number of lines in view mode
4	Current block mode (1 – line mode, 2 – column mode)

Com-
ments

Action	Default Keys
Cursor left	Cursor left
Cursor right	Cursor right
Cursor up	Cursor up
Cursor down	Cursor down
Enter	Enter
Delete the next character	Del
Delete the previous character	Backspace
Move to start of line	Home
Move up one page	Page up
Move down one page	Page down
Move to start of the editor	Shift-Home
Move to end of the editor	Shift-End
Move the selection block left one character	Shift-Cursor-Left
Move the selection block right one character	Shift-Cursor-Right
Move the selection block up one line	Shift-Cursor-Up
Move the selection block down one line	Shift-Cursor-Down
Cut the selection block to the clipboard	Ctrl-Del or Ctrl-X
Copy the selection block to the clipboard	Ctrl-Ins or Ctrl-C
Insert text from the clipboard	Shift-Ins or Ctrl-V

Example:

```
myEditor = MakeEditor();
ZBox(10, 110, 210, 10, 1);
Editor(10, 110, 210, 10 { Outline of editor },
       myEditor { Which editor to use },
       3 { Focus ID },
       Font("Courier" { Font name },
       0, 12 { Character set, height in points },
       0 { No rotation },
       7 { Weight (somewhat bold) },
       0, 1 { Not italicized, fixed pitch })
       infoArray { Information on the editor });
```

These statements create an editor that is displayed in a 100 x 200 area of the window in the upper left corner. This and other information will be stored in the array infoArray, whose data will be updated every time that one of the elements' values changes. Notice that the editor has been outlined by a dark blue box, which although not required, makes the boundaries of the editor obvious to the user.

Ellipse

Note: Deprecated. Do not use in new code.

Description: Draws an ellipse on the screen.

Returns: Nothing

Usage: Steady State only.

Function Groups: Graphics

Related to: Arc | Ball | Circle | GUIArc | GUIChord | GUIEllipse | GUIPie

Format: Ellipse(X, Y, XRadius, YRadius, Color, Width)

Parameters:

X

Required. Any numeric expression giving the X coordinate of the center of the ellipse on the screen.

Y

Required. Any numeric expression giving the Y

coordinate of the center of the ellipse on the screen.

XRadius

Required. Any numeric expression giving the radius of the ellipse along the X axis specified in units of X screen coordinates.

YRadius

Required. Any numeric expression giving the radius of the ellipse along the Y axis specified in units of Y screen coordinates.

Color

Required. A numeric expression giving the color of the ellipse.

Width

Required. Any numeric expression giving the width of the ellipse wall in units of X screen coordinates. The Width is always rounded to result in an odd number of pixels on the screen. The minimum width displayed will be 1 pixel.

Comments: This statement has been superseded by the GUIEllipse function and is maintained for backwards compatibility only. As of version 11, this is now drawn in the same z-order as other graphics, making it similar to the z-graphics functions.

Example:

```
Ellipse(400, 300 { Screen coordinates of ellipse center },
        200 { X radius in screen coordinates },
        100 { Y radius in screen coordinates },
        6 { Brown color },
        20 { Width of elliptical line in pixels });
```

This displays a brown ellipse in the middle of the screen.

Enable

Deprecated. Do not use in new code. (Alarm Manager module)

Description: Tell the Alarm Manager to enable an alarm. Use the SetEnable function for new code.

Returns: 0

Usage: Script Only.

Function Groups: Alarm

Related to: Register (Alarm Manager) | CurrentTime | SetEnable

Format: \AlarmManager\Enable(AlarmObject[, EventTime]);

Parameters:

AlarmObject

Required. The object value for the alarm that was passed to the Register subroutine that will be enabled.

EventTime

Optional. The time stamp to use when adding this event to the alarm lists. If invalid or not defined, the default is CurrentTime().

Comments: The Enable subroutine always returns "0".

EnableHelp

Description: Enables you to enable or disable help file activation when the F1 key is pressed.

Returns: Nothing

Usage: Script Only.

Function Groups: Help

Related to: Window | SetHelp

Format: EnableHelp(Enable, DefaultFileName)

Parameters:

Enable

Required. Controls whether or not F1 help is activated.

If Enable is set to zero, F1 help activation is disabled. If Enable is set to 1, F1 help activation is enabled.

DefaultFileName

Required. The name of the Windows help file to be launched when the user presses the F1 key (if F1 help is activated using the Enable parameter above), and if no other help context can be found.
This Windows help file is also the default help file that will be used when the user presses the Help button on the VTScada Application Manager (VAM).

Comments: VTScada calls EnableHelp during startup using the settings of the Setup.ini configuration file variables "F1DisableHelp" and "WEBHelp" respectively.

Example:

```
EnableHelp(PickValid(!F1DisableHelp, 1), WEBHelp);
```

Specify whether F1 help activation is enabled and the name of the default help file by setting the values of these variables in the Setup.ini configuration file.

Encode

(System Library)

Description: Processes a VTScada string using a configurable selection of compression, encryption, encoding and secure hashing.

Returns: String

Usage: Script Only.

Function Groups: Encryption

Related to: Decode | BlockEncrypt | Base64Encode | Hash | Pack

Format: \System\Encode(PlainValue[, PackDictionary, Compressed, Key, SaltLength, HashKey, Base64Encoded]);

Parameters:

PlainValue

Required. The information to be encoded. May be any VTScada value that can be packed.

PackDictionary

Optional dictionary. If present, the information will be packed. Refer to notes for the Pack function for further details about this parameter.

Compressed

Optional Boolean. Set TRUE if the value is to be compressed before possible encryption. No compression is done unless this value is specified as TRUE.

Key

Optional. Key to be used as a seed for encryption.

SaltLength

Optional numeric. Length of salt in bytes to use for encryption (0-64) Not relevant unless the Key parameter is also valid. Defaults to zero.

HashKey

Optional text. If valid, an SHA2-256 hash will be added to the end of the result to prevent tampering. This string is used to seed/salt the hash.

Base64Encoded

Optional Boolean. Set true if the result is to be Base64 encoded.

Comments: Note that if Base64 encoding is selected, the time required to encode and also to decode the information will increase by a factor of approximately ten.

Examples:

Encrypt

Description: The Encrypt function encrypts data. The algorithm used to encrypt the data is designated by the Key parameter. It is the VTScada analog of the CryptoAPI CryptEncrypt call.

Returns: Text

Usage: Script Only.

Function Groups: Cryptography

Related to: DeriveKey | Decrypt | Encrypt | ExportKey | GenerateKey | GetCryptoProvider | GetKeyParam | ImportKey | SetKeyParam

Format: Encrypt(Key, PlainText, Final [, Reserved, Flags, Error])

Parameters:

Key

Required. The handle to the key to use to encrypt the data.

PlainText

Required. A text string that contains the plain text to be encrypted.

Final

Required. A parameter that specifies whether this is the last section in a series being encrypted.
Final is set TRUE for the last or only block and FALSE if there are more blocks to be encrypted

Reserved

An optional parameter that should be set to 0. If omitted or invalid, then the value 0 is used.

Flags

Optional. Specifies the flags to be passed to CryptEncrypt. If omitted or invalid then the value 0 is used.
Refer to the Crypo API Encrypt function for the flag list.

Error

Optional. A variable in which the error code for the function is returned. It has the following meaning: (no default)

Error	Meaning
0	Key successfully imported.
1	Key, PlainText or Final parameters invalid.
x	Any other value is an error from CryptEncrypt.

Comments: The cipher text is returned as a text string. If an error occurs, the return value is invalid.

Example:

```
[
  PlainText1 = "abcdefghijklmnopqrstuvwxyz0123456789";
  CipherText1;
]
Init [
If 1 Main;
  [
    CipherText1 = Encrypt(Key3, PlainText1, 1, 0, 0);
  ]
]
```

ErrMessage

(ODBC Manager Library)

Description: Returns a text message for the error code handed to it as a parameter

Returns: Text message for the code provided.

Usage: Script or steady state.

Function Groups: ODBC

Related to:

Format: \ODBCManager\ErrMessage(ErrCode)

Parameters:

ErrCode

Required. A numeric error code for which you want to obtain the appropriate message.

Comments: This module is a member of the ODBCManager Library, and must therefore be prefaced by \ODBCManager\, as shown in "Format" above.

EvaluateAlarm

(Alarm Manager module)

Description: Passes a new value to an alarm, to be compared to the set-point.

Returns: Nothing

Usage: Script Only.

Function Groups: Alarm

Related to: Commission

Format: \AlarmManager\Evaluate(AlarmName, Value[, TimeStamp, Custom, Description])

Parameters:

AlarmName

Required text. The alarm name. Typically, the unique id of the alarm tag, or the tag containing built-in alarms.

Value

Required. The new value to be checked against the alarm setpoint.

TimeStamp

Optional UTC timestamp of the value. Defaults to the current time.

Custom

Optional structure of custom fields to be logged with

the event.

Description

Customized description, used if it differs from the description given to Commission.

Comments: EvaluateAlarm should be called every time your tag's Value changes.

Example:

Every tag that has commissioned an alarm should include the following line of code within its Refresh state:

```
 { Evaluate alarm condition }
\AlarmManager\EvaluateAlarm(AlarmName, Value, Timestamp);
```

Event

Note: Deprecated. Do not use in new code.

(Alarm Manager module)

Description: Tell the Alarm Manager when an alarm event occurs. This subroutine will cause an entry to be added to the log file without changing the alarm status.

Returns: 0

Usage: Script only.
May be used in optimized Tag Parameter Expressions.

Function Groups: Alarm

Related to: Register (Alarm Manager) (Alarm manager) | CurrentTime

Format: \AlarmManager\Event(AlarmObject[, EventTime]);

Parameters:

AlarmObject

Required. The object value for the alarm that was passed to the Register subroutine that will be enabled.

EventTime

Optional. The time stamp to use when adding this event to the alarm lists. If invalid or not defined, the default is CurrentTime().

Comments: The Event subroutine always returns "0".

Execute

Description: Executes a group of statements as a single entity in structures that would otherwise allow only one statement to be executed.

Returns: Nothing

Usage: Script Only.

Function Groups: Logic Control

Related to: Window | SetHelp

Format: Execute(Statement1 [, Statement2])

Parameters:

Statement1, Statement2, ... }

Required. Are any expression(s) to be executed. Any number of parameters may be used. Commas or semi-colons must separate the parameters.

Comments: This statement is typically used with a Case or IfElse statement..

Example:

```
If ZButton(10, 50, 110, 80, "Grid On", 1);
[
  { If module is not already launched, launch it },
  IfElse(! Valid(ModPtr),
  Execute( { Do a series of tasks }
    modPtr = ShowGrid(){Launched because of script},
    ySpace = ySpace,
    gridOn = 1
  ),
  { If module already launched, stop it }
  { else } Execute( { Do a different series of tasks },
  Slay(modPtr, 0),
  gridOn = 0
```

```
   ));
]
```

When this particular button is pressed, modPtr's validity is checked and if it is not valid, module ShowGrid is launched and some variables set, otherwise, ShowGrid is slain via modPtr and variable gridOn is set to 0. Notice how the use of the Execute function allows a whole series of tasks to be performed with only one condition check.

ExecuteQuery

(ODBC Manager Library)

Description: Called to send an SQL command to the server and get a reply back. This function is used as a general query tool and does not provide for the guaranteed eventual execution that the ExecuteQueryCached() function does.

Returns: 0 upon query execution starting. See the following comments.

Usage: Script Only.

Related to: ExecuteQueryCached

Format: \ODBCManager\ExecuteQuery(ErrPtr, CmdStr, DSN, UserName, Password, ResultPtr [, AttribPtr, ErrorMsgPtr, SQLStatePtr, ErrorCodePtr, ReFormat, TransObj, FormatBitField, dbType])

Parameters:

ErrPtr

Required. Pointer to an error. Always valid on completion. Set to 0 if the command succeeds.

CmdStr

Required. The SQL command to send to the database

DSN

Required. The name of the ODBC database in which to execute the command.

UserName

Required. The user name in the database for authentication. A null provided in this field will be passed to the database as a null string.

Password

Required. The password in the database for authentication. A null provided in this field will be passed to the database as a null string.

ResultPtr

Required. A pointer to the ODBC result data being passed back as the result of the query.

AttribPtr

Optional. A pointer to the ODBC attribute array being passed back from the result of the query. no default:

ErrorMsgPtr

Optional. A pointer to the ODBC error message being passed back. Will contain invalid if the command succeeds. no default:

SQLStatePtr

Optional. A pointer to the ODBC error state being passed back. Will contain invalid if the command succeeds. no default:

ErrorCodePtr

Optional. A pointer to ODBC error code being passed back. Will contain invalid if the command succeeds. no default:

ReFormat

Optional. A flag which, if set to true, causes the result set from the query to be reformatted from a Result [Column][Row] to Result[Row][Column]. This is potentially useful for reformatting multi-record "Select" queries to match your application requirements. no

default:

TransObj

Optional. An transaction object (as returned from the "Transaction()" function) to execute this query within. If a transaction is opened on this DSN and this value is not set, then the query will wait until the transaction has completed before being executed. If set, the query will be executed after all other outstanding ExecuteQuery() functions for the transaction on the DSN have executed. no default:

FormatBitField

Optional. Bitfield indicating whether values coming back from the query will be converted to their corresponding VTScada data types. Can be set on a type by type basis according to the following flags: (no default)

Value	Meaning
0	Convert numerics
1	Convert dates
2	Convert times
3	Convert timestamps

dbTyp

Optional numeric value, indicating the type of this DB connection.

DBType	Meaning
0	MS SQL
1	MS Access
2	Oracle
3	MySQL
4	SyBase

Comments: This module is a member of the ODBCManager Library, and must therefore be prefaced by \ODBCManager\, as shown in "Format" above.
This module MUST be called as a subroutine in a script.
This function acts as a shell to launch a query within the DSN within the ODBCManager library. Upon completion of the execution of the function, the query is still active. Completed execution of the query is indicated by a valid value set in the variable pointed to by parameter "ErrPtr". For this reason the variable referenced by "ErrPtr" MUST be invalidated before calling the function.

Example

```
Init [
  If 1 Wait Execution;
  [
    ErrPtr = Invalid();
    \ODBCManager\ExecuteQuery(&Err, "Select ID, TimeStamp, Data1 From
     Log_Table Order By TimeStamp", "MAIN LOGGER", User, Pass)
  ]
]

WaitExecution [
  If Err DisplayError;
  If !Err DisplayData;
]
```

ExecuteQueryCached

(ODBC Manager Library)

Description: Called to send an SQL command to the server and get a reply back. This module will cache the query locally if it fails & send it to the db after the next successful transaction with the db. This module was designed to be used for logging values that cannot be lost.

Returns: 0 upon query execution starting. See the following comments.

Usage: Script Only.

Related to: ExecuteQuery

Format: \ODBCManager\ExecuteQueryCached(ErrPtr, CmdStr,

DSN, UserName, Password[, BatchSize])

Parameters:

ErrPtr

Required. Pointer to an error. Always valid on completion. Set to 0 if the command succeeds.

CmdStr

Required. The SQL command to send to the database

DSN

Required. The name of the ODBC database in which to execute the command.

UserName

Required. The user name in the database for authentication. A null provided in this field will be passed to the database as a null string.

Password

Required. The password in the database for authentication. A null provided in this field will be passed to the database as a null string.

BatchSize

Optional. The number of array entries to send in one batch no default. Returns 0 upon query execution starting. See the following comments.

Comments:

This module is a member of the ODBCManager Library, and must therefore be prefaced by \ODBCManager\, as shown in "Format" above.
This module MUST be called as a subroutine in a script. Completed execution of the query is indicated by a valid value set in the variable pointed to by parameter "ErrPtr". For this reason the variable referenced by "ErrPtr" MUST be invalidated before calling the function.

Exp

Description: Returns the natural antilogarithm of a numeric expression.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Generic Math

Related to: Ln | Log | Pow

Format: Exp(X)

Parameters:

X

Required. Any numeric expression. The value must not be negative or the result will be invalid.

Comments: The function raises the constant e to the power of the parameter X. It is the complement of the Ln function.

Example:

```
a = Ln(78);
b = Exp(a);
```

In this example, b will be equal to 78.

ExportKey

Description: The ExportKey function exports a cryptographic key or a key pair from a CSP in a secure manner as a Key BLOB. It is the VTScada analog of the Crypto API ExportKey call.

Returns: Text

Usage: Script Only.

Function Groups: Cryptography

Related to: DeriveKey | Decrypt | Encrypt | GenerateKey | GetCryptoProvider | GetKeyParam | ImportKey | SetKeyParam

Format: ExportKey(Key, BlobType [, EncryptKey, Flags, Error])

Parameters:

Key

Required. The handle to the key which is to be exported.

BlobType

Required. A parameter specifying the type of key BLOB to be exported. Values are defined in WinCrypt.h

EncryptKey

Required. An optional parameter containing a Key handle for a key to be used to encrypt the exported key so that it may only be encrypted by the destination user. If omitted or invalid, then the value NULL is used.

Flags

Required. An optional parameter specifying the flags to be passed to CryptExportKey. If omitted or invalid then the value 0 is used. Flags values are defined in WinCrypt.h.

Error

Required. An optional variable in which the error code for the function is returned. It has the following meaning:

Error	Meaning
0	Key successfully exported.
1	Key or BlobType parameters invalid.
x	Any other value is an error from CryptExportKey.

Comments: The exported key is returned as a text string. If an error occurs, the return value is invalid.
EncryptKey is not required if BlobType is PUBLICKEYBLOB.

Example:

```
[
  PubKey1;
  Constant PUBLICKEYBLOB = 0x6;
]
Init [
  If 1 Main;
  [
    { Export the public key }
    PubKey1 = ExportKey(Key1, PUBLICKEYBLOB);
  ]
]
```

F Functions

The sections that follow identify all VTScada functions beginning with "F".

Fail

Modem Manager

Description: This subroutine advises the Modem Manager to abort and retry an established, outgoing call.

Usage: Script Only.

Related to:

Format: \ModemManager\Fail(Tag);

Parameters:

Tag

Any text expression that identifies the tag that originally requested the call.

Comments: None

FALSE

Description: For use in expressions that perform Boolean logic. Using "FALSE" will make your code easier to read than using "0".

Returns: With no parameters, returns the value, 0. If given a parameter, this function will return a 1 or 0 depending on whether the parameter evaluates to FALSE or TRUE. Always

returns 0 if the parameter is Invalid.

Usage: Script or steady state.

Function Groups: Logic Control

Related to: TRUE

Format: FALSE[(TestExpr)]

Parameters:

TestExpr

Optional. Any expression that evaluates to a 1 or 0 value. If no parameter is provided, then there is no need to include the parentheses.

Comments: This function exists to make your code more readable. It is equivalent to

```
PickValid(Cast(Parameter, 0) == 0, 0);
```

FFT

Description: Performs a fast Fourier transform between time and frequency domains.

Returns: Numeric

Usage: Script only.
May be used in optimized Tag Parameter Expressions.

Function Groups: Generic Math

Related to: ArrayOp1 | ArrayOp2

Format: FFT(Array, N, Operation)

Parameters:

Array

Required. The starting array element. Array may contain time domain samples, complex frequency samples, or amplitude/phase pair samples. Any data passed to FFT in Array will be overwritten by the res-

ulting transform. Arrays may be copied using ArrayOp2 if it is necessary to retain an original. There must be at least N elements in Array (partial transforms are not possible).

N

Required. Any numeric expression giving the number of elements in Array to transform. N must be a positive power of 2 in the range 4 to 4096.

Operation

Required. Any numeric expression giving the type of transform as follows:

Operation	Transform Type
0	Time samples to frequency complex
1	Time samples to frequency amplitude/phase
2	Time samples to frequency amplitude only
3	Frequency complex to time samples
4	Frequency amplitude/phase to time samples

Note that Operation 2 will set all phase elements to 0 to indicate a loss of phase information. This transform is not reversible because of this loss of phase information.

Comments: This statement is useful for performing frequency analysis on analog signals (such as motor frequency). In the time domain equal sampling intervals are required, equal frequency spacing in the frequency domain is required. The order of the time

domain samples is in chronological order.
The order of complex frequency data is...

```
real 0, real N/2, real 1, imaginary 1, real 2,
imaginary 2, ..., real N/2-1, imaginary N/2-1
```

Real 0 is the average (D.C.) component in the signal. The second element, real N/2, is the last real component because real 0 and real N/2 are always real (no imaginary component). Real 1 is the real amplitude of the lowest frequency, and imaginary 1 is the imaginary amplitude of the lowest frequency. Real N/2-1 and imaginary N/2-1 are the real and imaginary components of the second highest frequency.
The order of amplitude/phase pairs in Array is:

```
amplitude 0, amplitude 1, ..., amplitude N/2,
phase 1, phase 2, ..., phase N/2 - 1
```

Amplitude 0 is the average (D.C.) component in the signal. Amplitude 1 is the amplitude of the lowest frequency, amplitude N/2 is the amplitude of the highest frequency. Phase 1 is the phase angle (in radians) of the lowest frequency (the average doesn't have a phase, its frequency is 0). Phase N/2-1 is the phase angle of the second highest frequency.
Note that neither the average (D.C.) component nor the highest frequency have a phase angle or imaginary component. The highest frequency and the sampling is:
fc = 1/2D
Where fc is the critical (highest) frequency in Hertz, and D is the sampling interval in seconds. The lowest frequency, the number of samples, and the sampling interval are related by the following equa-

tion:
fL = 1/ND
Where fL is the lowest frequency in Hertz, N is the number of samples, and D is the sampling interval in seconds.
All other frequencies are multiplies of the lowest frequency fL.
Caution should be taken, as when VTScada executes a script, no other statements are updated. The FFT statement may take several seconds to execute, depending on the computer and the number of samples. The computation time goes up by N * log2N.

Example:

```
If 1 Main;
[
  FFT(samples[0], 512, 1);
]
```

Before this statement executes, samples[0] to samples[511] contain the equally spaced time samples of a signal, which will be overwritten by their amplitude/phase transform. After this statement executes, samples [0] will contain the average (D.C.) component of the signal. Samples[1] is the amplitude of fL, the lowest frequency component. Samples[256] is the amplitude of fc, the highest frequency component. Samples[257] will be the phase angle (in radians) of fL. Samples[511] will be the phase angle of fc.

FileDialogBox

Description:	Displays a threaded system common file dialog box.
Returns:	Numeric (1 = failure, 0 = success) In addition, see the Result parameter.
Usage:	Script Only.

Function Groups: File I/O, Graphics

Related to: FontDialog | PrintDialogBox | Dir

Threaded: Yes

Format: FileDialogBox(Save, FilterPattern, FilterDesc, File, Directory, Title, Extension, Result)

Parameters:

Save

Required. Any expression that evaluates to one of the following values

Save	Meaning
0	indicates that an "Open" dialog box is desired
Non-zero (positive)	indicates that a "Save" dialog box is required
-1	indicates that a "Browse for Folder" dialog is required

(please read Comments below if you require a "Browse for Folder" dialog).

FilterPattern

Required. Either a statically-declared array, or a semicolon separated list of wildcard file patterns for the file types that will be offered to the user. Defaults to an empty string if invalid.

FilterDesc

Required. Either a statically-declared array, or a semicolon separated list of text values that are the descriptions corresponding to the FilePattern values (e.g. "Text Files"). Defaults to an empty string if invalid.

File

Required. Any text expression giving the initial file

name for the dialog box.
In any 'Save' mode, an initial directory may be included as part of the file name. If the path is valid, the Directory parameter will be ignored. A known path alias may be provided in the form, {KnownPathAlias}.
Defaults to "" if invalid.

Directory

Required. Any text expression giving the initial directory for the dialog box. A known path alias may be provided in the form, :{KnownPathAlias} . (A table of known path aliases is provided in the Reference chapter).
Defaults to "" if invalid. See comments for more detail.

Title

Required. Any text expression giving the title of the window containing the dialog box. Defaults to "" if invalid.

Extension

Required. Any text expression giving the default file extension to use if the user does not specify one.
Defaults to "" if invalid.

Result

Required. A variable where the resulting file name, including its path, will be returned.

Comments: This statement displays a threaded system dialog box for opening or saving a file, depending on the value of the Save parameter, however, it does not actually perform the requested action, but simply displays the dialog. If successful, the Result parameter will be set to the full path and file name of the chosen file, or 0 if it fails or is canceled. It is the user's responsibility to act upon the value of Result and save or open the file by using such commands as FWrite or FRead.

In addition to the Result parameter, the function itself will return an error code to indicate whether the dialog was successfully opened. A "1" indicates failure to open while a "0" indicates success.
The Directory parameter is ignored for Open and Save operations if the File parameter contains a path. You can use this feature to define the initial directory, but return to the selected file if the user re-opens the dialog.
For example:

```
FileDialogBox(1, "", "", SelectedFile, Ini-
tialDirectory, ....);
```

SelectedFile can be initialized to "SomeFile.txt", in which case the InitialDirectory will be used. After the user has selected a file ("C:\AnotherFile.txt") , and reopens the dialog, then it will reopen to the path in SelectedFile. This avoids having to parse a user selection result (SelectedFile) into a filename and path to achieve the same effect.
For Save -1 (directory browser mode) the Directory parameter means something different. Directory defines the root of the selectable directories. For example, you can use this feature to restrict the user from selecting a directory outside their VTScada application folder. Passing the VTScada application path as Directory, means that operators would only be able to select that folder, or any sub-folders. This option can be used in conjunction with 'File' to restrict the selectable path and also have an initial selection.
If either the FilterPattern or FilterDesc parameters use dynamically allocated arrays (i.e. created using the New function), the dialog box will not open - these two parameters must use statically declared

arrays. If using a single text value instead, you may specify as many wildcard patterns as needed by adding a semi-colon separator between each: "*.BMP;*.JPG;*.PNG;*.TIF".

If the Save parameter is a negative value, indicating that a "Browse for Folder" dialog is required, the Directory and Title parameters must be set. Title will be displayed above the tree view control in the dialog box. This string can be used to specify instructions to the user. Directory can either be a string containing the starting root directory to begin browsing, or a CSIDL value to specify a special folder (a CSIDL is a number, the definitions of which can be found in the Platform SDK). The remaining parameters may be set to any valid value, including empty strings ("").
FTP sites may be browsed depending on the server's operating system. It will not work on Vista or Server 2008. FTP browsing is possible on Windows 7 and Server 2008 r2 .

Note: Within an Anywhere Client session, this function does nothing.

Example:

```
If ZButton(100, 176, 200, 146, "Browse", 1);
[
  FileDialogBox(0 { An "Open File" dialog box },
                "*.*" { All files is only option },
                "All Files" { Option label },
                "APPROOT.SRC" { Default file name },
                "C:\VTScada\APP" { Default path },
                "Find Application" { Window title },
                ".SRC" { Default extension },
                selectedApp { Chosen file });
]
```

When the button labeled "Browse" is pressed a file dialog box that shows all files (and directories) in the C:\VTS5\App directory will open. The title of the window will be "Find Application" and when a file is chosen, its path and name will be stored in selectedApp.

FileFind

Description: Performs a recursive search down through the directory tree structure and returns an array of matching file names.

Returns: Array

Usage: Script only.
May be used in optimized Tag Parameter Expressions.

Function Groups: File I/O

Related to: Dir | DriveInfo | FileDialogBox

Format: FileFind(Path, Attributes, Option)

Parameters: .

Path

Required. Any text expression that indicates the full path name where the search begins. This may include a wildcard spec, such as "*.DAT". The search progresses recursively down each sub-directory (depth-first).

Attributes

Required. Any numeric expression that gives the attributes to match on each file listed (files not matching these attributes won't be listed).

Attribute	Bit No.	Meaning
0	-	Files without attributes
8	3	All files (regardless of attributes)

Attribute	Bit No.	Meaning

Attribute	Bit No.	Meaning
1	0	Read only
2	1	Hidden
4	2	System
16	4	Sub-directory
32	5	Archive

Option

Required. Any numeric expression giving the type of text information to generate. The options are chosen by adding together all of the option numbers required. All information is placed in a single text string, separated by spaces. It will be stored in an element of the array that is created. The information is written from left to right, with lowest option numbers first.

Option	Bit No.	Generates
1	0	Short file name
2	1	Full path and file name
4	2	File size
8	3	File last modified date (in text)
16	4	File last modified time (in text)
32	5	File attributes (ADHRS)
64	6	File last modified date/time combination (in secs since Jan. 1, 1970)
128	7	File creation date (in text)
256	8	File creation time (in text)
512	9	File creation date/time combination (in secs since Jan. 1, 1970)

The attributes returned as a result of bit 5 being set are printed as the capital letters A (archive), D (subdirectory), H (hidden), R (read-only), and S (system).

Comments: This function returns an array of text values. As the function does a recursive search through the directory tree, the contents of each sub-directory are added to the array in reverse-alphabetic order before the function moves on to the next directory.
Each text value contains the information specified by Option for each file which matches both Path and Attributes. If no files are found, the return value will be set to invalid. Notice that the only difference between this function and the Dir function is that Dir searches in the immediate directory only, while this function looks down through the whole directory tree.

Example:

```
If 1 Main;
[
  fileData = FileFind("C:\VTS5\App\*.DAT", 8, 8);
]
```

This statement will find all files with the extension "*.DAT" in the given directory regardless of their attributes. The array fileData will contain each file's full path and file name as well as its size and date.

FileRootModule

Description: Parses the document file that contains the given module to find the root module in that file. Returns the module value of the root module.

Returns: Pointer

Usage: Script Only.

Function Groups: Compilation and On-Line Modifications, Advanced Module, File I/O

Related to: SystemSelf

Format: FileRootModule(Module)

Parameters:

Module

Required. Any expression for the module.

Comments: The returned module value from this statement may be the parent or some ancestor of Module.

Example:

```
modRoot = FileRootModule(Self());
```

This function returns a pointer to the root module of the current module.

FileSize

Description: Returns the size of a disk file in bytes.

Returns: Numeric

Usage: Script only.
May be used in optimized Tag Parameter Expressions.

Function Groups: File I/O

Related to: FRead | FWrite | GetStreamLength

Format: FileSize(FileName)

Parameters:

FileName

Required. Any text expression that gives the path, file name, and extension of the file. A Known Path Aliases for File-Related Functions may be provided in the form, :{KnownPathAlias}.

Comments: This function returns the size of the file in bytes. If the return value is invalid, the file could not be found.

Example:

```
If ! Valid(size);
[
  size = FileSize("C:\Log\Samples.DAT")
]
```

Size is set to the size in bytes of the file indicated.

FileStream

Description: Returns a stream attached to a disk file or printer, and is suitable for use in SWrite.

Returns: Stream

Usage: Script Only.

Function Groups: File I/O, Stream and Socket

Related to: BlockWrite | BuffStream | ClientSocket | CloseStream | GetStreamLength | PipeStream | PrintDialogBox | ServerSocket | SRead | StreamEnd | SWrite

Format: FileStream(FileName [, PrintFlag, CompletionVar, Flags])

Parameters:

FileName

Required. Any text specification for a file or printer SWrite. A known path alias may be provided in the form, :{KnownPathAlias}.

PrintFlag

An optional parameter that is any logical expression and must be set to true (non-0) if the stream is printer-based so that special handling and error checking for printer output will be provided.
If this parameter is omitted, the default value of false (0) is used (i.e. a file-based stream is assumed).

CompletionVar

An optional variable that will be initially set Invalid.

When the stream is closed it will be incrementally set to an integer in the range 0 to 100. This integer indicates the percentage completion of writing the session-aware stream contents to the VIC.

When the value of this variable is set to exactly 100, the stream has been completely written to the VIC file system. This can be used by the programmer to indicate the progress in writing large files to a VIC over a relatively slow communication link.

This variable can also be provided on non-session-aware FileStreams, where it will remain untouched by any file operations. Its presence does not mark the FileStream as being session-aware.

Flags

An optional parameter that may take one of three values.
If absent or Invalid, the FileStream will not be session-aware (i.e. all operations will be on the local file system.
If valid, the FileStream is session-aware, and if the file does not exist on the VIC file system, a zero-length file will be created and opened for read-write access.
If the file on the VIC exists, setting this parameter to zero will cause it to be initially opened for read-only access. Read-write access is only obtained when you make the first write to the stream.
If the file exists on the VIC, setting Flags to 1 will cause the file to be opened for read-write access, and truncated to zero length.

Comments: If the file designated by FileName does not exist, it will be created once it has been written to (by SWrite/BlockWrite for example). The file pointer returned by the function will

still be valid. If the file exists but has its read-only attribute set, the stream may only be read from, not written to.
If you want to be certain that the file exists, then manually use FWrite(File, 2, 0, ""); before opening the stream with FileStream.
If the PrintFlag parameter is set, even though a printer is not being used (i.e. FileName indicates a certain file), it will not have any effect overall except that it may impose a slight performance penalty.

Session-aware Streams:

Session-aware streams can be defined simply as streams opened or created by a FileStream statement that can refer to files system resources on the server, or on a VIC. In the latter case, the stream is referred to as a "remote" stream. Not all FileStream statements are session-aware. As the programmer, you must decide whether to make a FileStream session-aware.

To make a FileStream session-aware, you supply the additional, optional parameters, CompletionVar and Flags, to indicate that the stream is a session-aware stream. Without these additional parameters (or with the additional parameters set to certain values), the FileStream is treated just as a regular FileStream.

Having marked a FileStream statement as session-aware, the FileStream will be a remote stream only if it is running in the context of a VIC session. In other words, the module instance running the FileStream statement has an ultimate caller module instance that is the root instance for a VIC session.

Note: A VIC session differs from a DisplayManager session. A DisplayManager session may be running

in the singleton server session, or it may be running in one of N VIC sessions. The concept of VIC session is something that the VTScada engine understands, whereas the concept of a DisplayManager session is something that the VTScada layer understands.

When a FileStream is remote, the FileName parameter is provided relative to the VIC. For example, "C:\Temp\XX.txt" refers to a file on the C drive of the VIC. The FileDialogBox statement may be used to allow the VIC user to identify a resource accessible to the VIC, and provide a suitable text string to pump into FileStream's FileName parameter.
If used in an Anywhere Client, calling FileStream with the Flags set so as to be session-aware causes FileStream to do nothing and return Invalid.

Blocking:

It is necessary to block the execution of VTScada script code at clearly defined points when using a remote FileStream. Blocking script execution blocks only the executing interpreter thread (i.e. it does not cause VTScada to stop executing statements in other threads).
The points at which VTScada script code execution will be blocked are...

- During execution of a FileStream statement that refers to a remote stream. Execution will block until the VIC can tell the server whether or not the specified FileName parameter is legal, and if it is, the size of the file. This allows a GetStreamLength statement immediately following the FileStream statement to return the size without blocking. It also allows the engine to initialize a cache for the remote stream.
- On the first write to the remote stream, if the remote stream is open as read-only. This can be a result of a

number of statements, such as BlockWrite or SWrite. The block is obtained simply to learn whether or not read-write access can be obtained, and get a "write-lock" on the remote stream. This lock prevents other processes from accessing the stream content until VTScada closes it.

- On all writes to a remote stream when the write overwrites the content in the remote stream, and the remote stream content for that file offset is not cached locally on the server. This is not strictly necessary, and is a product of the current caching algorithm that operates on fixed size blocks. If this becomes restrictive, a smarter caching algorithm can avoid this block.
- On all reads to a remote stream when the content for the specified file offset is not cached locally on the server.

File Content Transfer: Remote stream content is cached on the server in fixed size blocks. Only those blocks that have been read or written are held on the server. Any changed blocks are only written to the remote stream when the stream is closed, either explicitly with a CloseStream statement, or implicitly by invalidating the last reference to the remote stream. The engine does not block interpreter threads while this operation is performed.

All transfer to the VIC is performed on the "low-priority" channel within the VIC connection to the server. Graphics, user-input and control actions are all performed using the "high-priority" channel. Presently the only other transfer that takes place over the low-priority channel is image transfer. This means that operator actions and graphical representations are largely unaffected by stream transfer operations. Because only one channel is used for stream transfer, all stream transactions are serialized within that

channel, meaning that an attempt to open a stream that the server has just closed will be queued behind the last write to the stream, ensuring that you cannot read stale data from the remote stream.
The CompletionVar parameter to the session-aware FileStream statement may be used to monitor the progress of large transfers, and provide operator feedback. It is important to remember that because graphics use a higher priority channel than the stream, the operator will be able to see any progress notification that you display.

Caching Considerations:

The present implementation of remote streams uses a fixed block size cache (4 kb) on the server. If an attempt is made either to write a partial file block that is not in the server cache, but is existent in the remote stream. Or, to read from a block that is not in the server cache but is existent in the remote stream, the calling script code will be blocked until the server has recovered the stream data for that block from the VIC. Dependent upon the connection speed and quality, this may take a long time (in computer terms).
Writes of an entire block that is not cached does not cause the server to read data from the VIC.
Once a block has been cached on the server, it is not re-read from the VIC.
This algorithm provides a balance between complexity and performance, principally designed to give good performance when creating a file from scratch and writing it to the VIC, while providing some caching benefit for random access use of the remote stream.

Example:

```
If ! Valid(stream);
[
  stream = FileStream("C:\Recipe\Glue.DAT");
]
```

This statement will set stream to the stream of the file indicated.

```
If MatchKeys(1, "p");
[
  printStream = FileStream("\\ServerName\PName", 1);
]
```

This will cause printStream to be created as a printer based stream pointing to the printer called PName on the server called ServerName.

Filter

Description: Sets the value of one array element to invalid if the corresponding value in another array element is invalid.

Returns: Nothing

Usage: Script only.
May be used in optimized Tag Parameter Expressions.

Function Groups: Array

Related to: ArrayOp1 | ArrayOp2 | FiltHigh | FiltLow

Format: Filter(Array1Elem, Array2Elem, N)

Parameters:

Array1Elem

Required. Any array element giving the starting point for the array conversion in the destination array. The subscript for the array may be any numeric expression. If processing a multidimensional array, the usual rules apply to decide which dimension should be used.

Array2Elem

Required. Any array element giving the starting point in the reference array (this array is not altered in any way). The subscript for the array may be any numeric expression. If processing a multidimensional array, the usual rules apply to decide which dimension should be used.

N

Required. Any numeric expression giving the number

of array elements to process starting at the element given by the first parameters. If this parameter is greater than either dimension of the arrays, the number of points used will be the smaller array dimension.

Comments: This statement is useful together with either the FiltLow or FiltHigh statements which would typically be executed on the first array before Filter is executed.

Example:

Assume that there exists 2 arrays both of whom have subscripts starting at 0, such that x = { 1, 2, Invalid, 4 } and Y = { 2.3, Invalid, 2.4, Invalid }

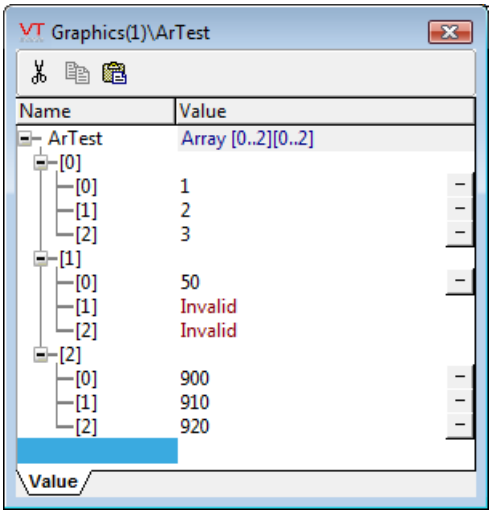
```
If changeArray;
[
  Filter(x[0] { Start of array to be changed },
       y[0] { Start of reference array },
        4 { Number of elements to process });
  changeArray = 0;
]
```

If the variable changeArray is set to true, this will result in array y remaining unchanged and array x being changed to x = {1, Invalid, Invalid, Invalid }

Example 2:

```
  IF Watch(1);
  [
    ArTest = new(3,3);
    ArTest2 = new(3,3);

    ArTest[0][0] = 1;
    ArTest[0][1] = 2;
    ArTest[0][2] = 3;
    ArTest[1][0] = 50;
    ArTest[1][1] = 60;
    ArTest[1][2] = 70;
    ArTest[2][0] = 900;
    ArTest[2][1] = 910;
    ArTest[2][2] = 920;

    ArTest2[0][0] = 1;
    ArTest2[0][2] = 3;
    ArTest2[1][0] = 50;

    Filter(ArTest[1], ArTest2[1], 3);
]
```

After filtering, ArTest will look like:



FiltHigh

Description: Sets the values in an array sub-range that fall above a specified upper limit to a new value.

Returns: Nothing

Usage: Script only.
May be used in optimized Tag Parameter Expressions.

Function Groups: Array

Related to: ArrayOp1 | ArrayOp2 | Filter | FiltLow

Format: FiltHigh(ArrayElem, N, Limit, Value)

Parameters:

ArrayElem

Required. Any array element giving the starting point in the array for the search. The subscript for the array may be any numeric expression. If processing a multidimensional, the usual rules apply to decide which dimension should be used.

N

Required. Any numeric expression giving the number of array elements to use, starting at the element given

by the first parameter. If N extends past the upper bound of the lowest array dimension, this computation will "wrap-around" and resume at element 0, until N elements have been processed.

Limit

Required. Any numeric expression giving the upper cutoff value for the array elements. Any array elements in the range that are strictly greater than this value are set to the Value parameter.

Value

Required. Any numeric expression giving the new value to set the array elements that fall above the Limit parameter. The function still continues if this parameter is invalid, in which case the new values become invalid.

Comments: The statement is useful for setting array elements above a limit to a maximum value or to invalid.

Example:

Assume that there exists an array whose subscripts start at 0, such that x = { 2, 1, invalid, 10 }

```
If ChangeArray;
[
  FiltHigh(x[0] { Start of array },
          4 { Number of elements to process },
          2 { Max value in the array },
          Invalid { Default value if over the limit });
  changeArray = 0;
]
```

If the variable changeArray is set to true, x will be changed to x = {2, 1, Invalid, Invalid }

FiltLow

Description: Sets the values in an array sub-range that fall below a specified lower limit to a new value.

Returns: Nothing

Usage: Script only.
May be used in optimized Tag Parameter Expressions.

Function Groups: Array

Related to: ArrayOp1 | ArrayOp2 | Filter | FiltHigh

Format: FiltLow(ArrayElem, N, Limit, Value)

Parameters:

ArrayElem

Required. Any array element giving the starting point in the array for the search. The subscript for the array may be any numeric expression.
If processing a multidimensional array, the usual rules apply to decide which dimension should be used.

N

Required. Any numeric expression giving the number of array elements to use, starting at the element given by the first parameter. If N extends past the upper bound of the lowest array dimension, this computation will "wrap-around" and resume at element 0, until N elements have been processed.

Limit

Required. Any numeric expression giving the lower cutoff value for the array elements. Any array elements in the range that are strictly less than this value are set to the Value parameter.

Value

Required. Any numeric expression giving the new value to set the array elements that fall below the Limit parameter. The function still continues if this parameter is invalid, in which case the new values become invalid.

Comments: The statement is useful for setting array elements below a limit to a minimum value or to invalid.

Example:

Assume that there exists an array whose subscripts start at 0, such that x = { 2, 1, Invalid, 10 }

```
If changeArray;
[
  FiltLow(x[0] { Start of array },
         4 { Number of elements to process },
         2 { Min value in the array },
         Invalid { Default value if under the limit });
  changeArray = 0;
]
```

If the variable changeArray is set to true, x will be changed to x = {2, Invalid, Invalid, 10 }

FindAction

Description: Returns an action from the list of actions in a state.

Returns: Pointer

Usage: Script Only.

Function Groups: Compilation and On-Line Modifications, State

Related to:

Format: FindAction(Action, Mode)

Parameters:

Action

Required. Any expression that gives a code value. If the code value represents a module and state, the first action will be returned.
If the code value represents an action or statement, the action returned will depend on Mode.

Mode

Required. Any numeric expression for the mode. The mode is described by the following table:

Value	Mode
-1	Previous action
0	Return Action parameter
1	Next action

If Mode is 1, and Action is the last action in the state, or if Mode is -1, and Action is the first predicate in the state, the return value is invalid.

Comments: This function is used to step through the actions in a state.

FindModem

Description This subroutine returns a pointer to one of the Modem Manager's own internal modem objects. This pointer may then be used to access public, read-only properties for display purposes.

Usage Script Only.

Related to:

Format \ModemManager\FindModem(ModemName);

Parameters

ModemName

Any text expression that identifies the required modem. This will be the name property of a modem tag.

Comments FindModem will return an object pointer for a given modem tag name. This object refers to a Modem Manager internal object, which provides access to several read-only properties. These are:

Property	Description

Workstation	Name of the workstation where the modem exists
FriendlyName	Modem name as displayed in the Windows Control Panel's Phone and Modems option (or Modems option)
Failed	. Refer to Call Progress and Error Codes. in the VTScada Programmer's Guide.
OffHook	TRUE while a call or call setup is in progress
Connecting	TRUE while call setup is in progress

FindVariable

Description: Searches for a variable by text name and returns a variable value.

Returns: Varies

Usage: Script Only.

Function Groups: Compilation and On-Line Modifications, Variable

Related to: AddVariable | DeleteVariable | MakeNonShared | MakeNonPersistent | MakePersistent | MakeShared | SetDefault | SetVariableClass | SetVariableText | ListVars

Format: FindVariable(Name, Module, Reserved, Global)

Parameters:

Name

Required. Any text expression that gives the name of the variable.

Module

Required. Any expression for the module where the search begins.

Reserved n/a

Reserved for future use, set to 0.

Global

Required. Any logical expression. If true (non-0), the search will continue to parent modules if they exist and the variable isn't found. If false (0), only Module will be searched.

Comments: This function returns invalid if the variable is not found. FindVariable can be used on the left side of an equals sign (=) to allow shared variables to be set when there are no instances of a module running.

Example:

```
If ! Valid(ptr);
[
  ptr = Launch(FindVariable("Grid", Self(), 0, 1 { Launchee }),
               FindVariable("Draw", Self(), 0, 1 { Parent }),
               FindVariable("Draw", Self(), 0, 1 { Caller }),
               xSpace, ySpace { Parms });
]
```

This statement launches module Grid with its parameters xSpace and ySpace as if it were a child of module Draw and had been called by Draw. If Draw had been assigned an object pointer at the time it was called, the second and third FindVariable statements would be unnecessary.

```
If Valid(checkVar);
[
  IfThen(!Valid(FindVariable(checkVar { Var to look for },
                             Self() { Check this module },
                             0 { Reserved },
                             0 { Only this module })),
         message = "Variable does not exist!");
]
```

This statement checks to see if a certain variable exists by looking for it by name in the current module only. If it does not exist, an error message is set.

FirstState

Description: Sets the first state in a module.

Returns: Nothing

Usage: Script Only.

Function Groups: Compilation and On-Line Modifications, State

Related to:

Format: FirstState(State)

Parameters:

State

Required. Any expression for the code value of the new first state.

Comments: Given that a module starts in its first state, this function will set which state within a module will considered to be first.

FitOffset

Description Linear regression offset. This function returns the offset or Y intercept of the least square curve fit of data in a pair of arrays.

Returns Numeric

Usage Script or steady state.

Function Groups Generic Math

Related to: FitR2 | FitSlope

Format FitOffset(XArrayElem, YArrayElem, N)

Parameters

XArrayElem

Required. Any array element giving the starting point in the array of X coordinates of the input data set. The subscript for the array may be any numeric expression.

If processing a multidimensional array, the usual rules apply to decide which dimension should be used.

YArrayElem

Required. Any array element giving the starting point

in the array of Y coordinates of the input data set. The subscript for the array may be any numeric expression.

If processing a multidimensional array, the usual rules apply to decide which dimension should be used.

N

Required. Any numeric expression giving the number of data points to use from the arrays given by the first two parameters.

If N extends past the upper bound of the lowest array dimension, this computation will "wrap-around" and resume at element 0, until N elements have been processed.

Comments If an element of either array is invalid, then that X-Y pair is not included in the computation. If the number of valid data points is less than 2, the function returns an invalid value. Note that XarrayElem and YarrayElem are not necessarily the same array element number. This function is used in conjunction with the FitSlope function.

Example:

Assume that 2 arrays exist such that x = {0, 1, 2, 3} and y = {1, 3, 5, 7 }, and both arrays' subscripts start at 0

```
intercept = FitOffset(x[0] { Starting X element },
                      y[0] { Starting Y element },
                      4 { Number of elements to process });
```

The variable intercept will be set to 1.

FitR2

Description: Returns the coefficient of determination (i.e. r2) for a linear curve fit. This number gives a measure of how accurate the curve fit is.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Generic Math

Related to: FitOffset | FitSlope

Format: FitR2(XarrayElem, YarrayElem, N)

Parameters:

XArrayElem

Required. Any array element giving the starting point in the array of X coordinates of the input data set. The subscript for the array may be any numeric expression.

If processing a multidimensional array, the usual rules apply to decide which dimension should be used.

YArrayElem

Required. Any array element giving the starting point in the array of Y coordinates of the input data set. The subscript for the array may be any numeric expression.

If processing a multidimensional, the usual rules apply to decide which dimension should be used.

N

Required. Any numeric expression giving the number of data points to use from the arrays given by the first two parameters.

If N extends past the upper bound of the lowest array dimension, this computation will "wrap-around" and resume at element 0, until N elements have been processed.

Comments: This function returns a number indicating how close a fit the data are to a line. If an element of either array is invalid, then that X-Y pair is not included in the computation. If the result is 0, there is no linear relationship at all between the array of X values and the array of Y values.

If the result is 1, the fit is perfect. The result may be any value in the range of 0 to 1.
This function can be used in conjunction with the other linear regression functions.

Example:
Assume that 2 arrays exist such that x = {0, 1, 2, 3} and y = {1, 3, 5, 7 }, and both arrays' subscripts start at 0

```
determination = FitR2(x[0] { Starting X element },
                      y[0] { Starting Y element },
                      4 { Number of elements to process });
```

The value of determination will be set to 1 (a perfect fit).

FitSlope

Description: Linear regression slope. This function returns the slope of the least square curve fit of data in a pair of arrays.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Generic Math

Related to: FitOffset | FitSlope

Format: FitSlope(XarrayElem, YarrayElem, N)

Parameters:

XArrayElem

Required. Any array element giving the starting point in the array of X coordinates of the input data set. The subscript for the array may be any numeric expression.
If processing a multidimensional array, the usual rules apply to decide which dimension should be used.

YArrayElem

Required. Any array element giving the starting point

in the array of Y coordinates of the input data set. The subscript for the array may be any numeric expression.
If processing a multidimensional array, the usual rules apply to decide which dimension should be used.

N

Required. Any numeric expression giving the number of data points to use from the arrays given by the first two parameters.
If N extends past the upper bound of the lowest array dimension, this computation will "wrap-around" and resume at element 0, until N elements have been processed.

Comments: If an element of either array is invalid, then that X-Y pair is not included in the computation. If the number of valid data points is less than 2, the function returns an invalid value. Note that XArrayElem and YArrayElem are not necessarily the same array element number. This function is used in conjunction with the FitOffset function.

Example:

Assume that 2 arrays exist such that x = {0, 1, 2, 3} and y = {1, 3, 5, 7 }, and both arrays' subscripts start at 0.

```
slope = FitSlope(x[0] { Starting X element },
                 y[0] { Starting Y element },
                 4 { Number of elements to process });
```

The variable slope will be set to 2.

Flush

Description: Pushes the data in all software caches associated with a FileStream directly to the physical media.

Returns: Nothing (see parameters)

Usage: Script Only.

Function Groups: Stream and Socket

Related to: Diff | CloseStream

Threaded: Yes

Format: Flush(Stream, CompletionCounter, Success);

Parameters:

Stream

Required. Any expression that resolves to the stream that is to be persisted to the physical media. This must be a FileStream. Other stream types do nothing in response to a Flush call.

CompletionCounter

Required. Any expression that resolves to a variable containing a numeric value or Invalid. If a numeric variable, the value will be incremented at the instant that Flush is called. It will then be decremented after the stream has been flushed. The same variable can be used to monitor any number of simultaneous, asynchronous Flush operations.
If this parameter is set to Invalid then the Flush operation will be performed synchronously and the function won't return until the stream is flushed.

Success

Required. Any expression that resolves to a variable containing a numeric value or Invalid. If a numeric value, that value will be set once the stream has been flushed to indicate the success of the operation. A value of TRUE (1) indicates success while FALSE (0) indicates failure. If set to invalid then the success of the operation is not reported.

Comments: The operation is bound to the speed of the physical media and can be slow.

This function is used to ensure that all preceding operations on a FileStream have been completed *on the physical media* before the operation completes. All file operations in VTScada are subject to the mediation of the file cache (a part of the Operating System that serves to speed up file access) which can have a reliability cost when the cache or the media are disrupted.

This function allows the caller to momentarily opt out of the file cache, ensuring that a file is in the expected state while exposing the caller to the performance limitations of the physical device. As this can be a very slow operation it is performed asynchronously, with the caller being informed once the operation completes. Note that an asynchronous flush can be prevented by the CloseStream function if that results in the file being closed before the flush completes.

There are two types of asynchronous operation available, depending upon whether the CompletionCounter value has been set to a variable containing a valid or invalid value. If invalid, the thread upon which the call was made is blocked until the operation completes, but other threads are allowed to run.

CompletionCounter should not be set invalid if Flush is called in a CriticalSection as this will cause all threads to await the flush operation.

If CompletionCounter is set to a valid numeric value then the flush operation will occur independently of the calling thread, which will continue executing immediately. This mechanism is identical to that

used by the Diff function, and the CompletionCounter parameters of the two operations can be shared.
If the Success parameter is provided and has not been set to 1 by the time the operation completes then it should be assumed that the file has not been persisted, and in fact may not be persist-able. This can happen if there is something wrong with the Stream parameter, or if either the OS file cache or the physical media are damaged. Flushing a file that has nothing to write will set this parameter to TRUE (1).

FlushCache

(ODBC Manager Library)

Description: Forces a flush of a log file for a specified DSN. Returns an error pointer to indicate success or failure.

Returns: Numeric

Usage: Script Only.

Related to:

Format: \ODBCManager\FlushCache(DSN, UserName, Password, ErrorPtr)

Parameters:

DSN

Required. The data source name of the ODBC database for which to flush the cache.

UserName

Required The user name in the database for authentication. A null provided in this field will be passed to the database as a null string.

Password

Required. The password in the database for authentication. A null provided in this field will be passed to the database as a null string.

ErrPtr

Required. Pointer to an error. Always valid on completion. Set to 0 if the command succeeds.

Returns:0 upon completion.

Comments: This module is a member of the ODBCManager Library, and must therefore be prefaced by \ODBCManager\, as shown in "Format" above.

FocusID

Description: Returns the focus ID of the object in a window that has the input focus.

Returns: Numeric

Usage: Script Only.

Function Groups: Window, Graphics, Keyboard

Related to: NextFocusID

Format: FocusID(Window)

Parameters:

Window

Required. Any object value that will specify the instance of a window.

Comments: If the window is inactive, this function will return the focus ID of the object that will receive the input focus when the window becomes active.

Example:

```
ZButton(10, 100, 110, 130, "OK", 1);
If LocSwitch() == 4 { Left mouse button pressed };
```

```
[
  IfThen(FocusID(Self()) != 1 { Focus leaves the OK button },
       NextFocusID(Self(),1) { Return focus to OK button });
]
```

This set of statements ensures that after any input by the mouse (i.e. clicking of the left mouse button), the focus will return to the "OK" button.

Folder

(System Library)

Description: Draws a tabbed folder dialog.

Returns: Nothing

Usage: Steady State only.

Function Groups: Graphics

Related to: DialogInitPos

Format: \System\Folder(X1, Y1, X2, Y2, Labels, Selected)

Parameters:

X1

Required. The coordinate of the left side of the folder.

Y1

Required. The coordinate of the bottom of the folder.

X2

Required. The coordinate of the right side of the folder.

Y2

Required. The coordinate of the top of the folder.

Labels

Required. A one dimensional array of labels to be assigned as the labels for each of the folder's tabs.

Selected

Required. The selected tab. Selected defaults to 0

(indicating the first tab).

TabHeight

Numeric feedback parameter, telling the caller the height of the tabs, or the total height when there are multiple rows of tabs.

Comments: This module is a member of the System Library, and must therefore be prefaced by \System\ as shown in the example above (see Format above). If the application you are developing is a script application, the System variable must be declared in AppRoot.src, and need not be prefaced by a backslash in the function call.

Example:

```
FolderOn [
 If Valid(FolderOff) FolderOff;
  \System\Folder(10, 10, WIDTH - 10, HEIGHT - BUTTONH - 20,
                 TabNames, Current);
 ]
```

where Width is a variable indicating the window width, Height is a variable indicating the window height, ButtonH is a constant with a value of 30, TabNames is a variable indicating the labels to be displayed on each tab, and Current is a variable indicating the current tab selected on the dialog should be the selected tab.

Font

Description: Returns a font value.

Returns: Font

Usage: Steady State only.

Function Groups: Graphics

Related to: FontDialog | GUIText | ZText

Format: Font(Name, CharSet, Height, Rotation, Weight, Italic, Fixed)

Parameters:

Name

Required. Any text expression that gives the name of the font. This must be the same as the Microsoft Windows™ name for the font. For example, "ARIAL".

CharSet

Required. Any numeric expression giving the Font Character Sets for this font.

If you are uncertain as to a valid value, set CharSet to 0, thereby obtaining English characters.

The keyword, DEFAULT_CHARSET may be used, auto-selecting the character set based on the configured Windows locale. OEM_CHARSET will do the same, but will use the DOS equivalent.

Height

Required. Any numeric expression giving the height of the font in points.

Rotation

Required. Any numeric expression giving the rotation of each character in degrees.

Weight

Required. Any numeric expression giving the weight of the font. Larger numbers give a more bold appearance. The range is 0 to 9.

Italic

Required. Any logical expression. If true (non-0), the italicized version of the font is used. If false (0), the normal version is used.

Fixed

Required. Any logical expression. If true (non-0), all of

the characters used by the font will be the same width and height, that of the largest character. If false (0), and the font is a proportional or a true type font, then the characters may have different sizes.

Comments: This function is for use in layered graphics statements that display text. A good idea is to use variables for font parameters in the layered graphics statements. If it is desired to change the font later, it need only be changed at one place – where the assignment is made to the variable. This also promotes a consistent use of fonts.
Parameters are suggestions. Substitutions will be made if the parameters describe a font that cannot be found.
Supported font types include TRUEType, OpenType and PostScript CFF.

Example:

```
inputFont = Font("ARIAL" { Font name },
                 0 { Character set },
                 14 { Height in points },
                 0 { Rotation },
                 5 { Weight - somewhat bold },
                 0 { Not italic },
                 0 { Non-fixed });
```

FontDialog

Description: Displays a threaded system common font dialog box.

Returns: Error code

Usage: Script Only.

Function Groups: Graphics

Related to: Font | PrintDialogBox

Threaded: Yes

Format: FontDialog(Name, CharSet, Height, Rotation, Weight, Italic, Fixed [, Display, Result])

Parameters:

Name

Required. Any text expression which gives the name of the font. This must be the same as the Microsoft Windows™ name for the font. For example, "MS Sans Serif".

CharSet

Required. Any numeric expression giving the Font Character Sets for this font.
If you are uncertain as to a valid value, set CharSet to 0, thereby obtaining English characters.
The keyword, DEFAULT_CHARSET may be used, auto-selecting the character set based on the configured Windows locale. OEM_CHARSET will do the same, but will use the DOS equivalent.

Height

Required. Any numeric expression giving the height of the font in points.

Rotation

Required. Any numeric expression giving the rotation of each character in degrees.

Weight

Required. Any numeric expression giving the weight of the font. Larger numbers give a more bold appearance. The range is 0 to 9.

Italic

Required. Any logical expression. If true (non-0), the italicized version of the font is used. If false (0), the normal version is used.

Fixed

Required. Any logical expression. If true (non-0), all of the characters used by the font will be the same width and height, that of the largest character. If false (0), and the font is a proportional or a true type font they may have different sized characters.

Display

An optional parameter that gives a list of fonts to display. Display can be set to one of the following values.

Display	Font List
0	All fonts
1	Screen fonts only
2	Printer fonts only

If the value is invalid, all fonts will be displayed. If out of range, then the function call is not valid.

Result

An optional parameter that is a variable whose value will be set to 1 if the OK button on the font dialog is clicked, and 0 if the Cancel button on the font dialog is clicked.

Comments: The first seven parameters are originally read in to set the default values for the dialog box and if they are VTScada variables, they will be set to the attributes for the font chosen by the user in the dialog box.

In addition to the Result parameter, the function itself will return an error code to indicate whether the dialog was successfully opened. A "1" indicates failure to open while a "0" indicates success.

Note: Within an Anywhere Client session, this function does nothing.

Example:

```
[
  newFont;
  name = "Century Gothic";
  charSet = 0;
  ht = 14;
  rotate = 0;
  wt = 5;
  italic = 1;
  fixed = 1;
  wasSet = 0;
]
Main [
  { Display the chosen font }
  ZText(100, 100, Concat("A ", name, " Font", 12, newFont);
  newFont = Font(name, charSet, ht, rotate, wt, italic, fixed);
  { Set the font }
  If ZButton(200, 10, 300, 40, "Set Font", 1) &&
    Valid(wasSet);
  [
    wasSet = Invalid;
    FontDialog(name { Font name },
              charSet { Character set },
               ht { Height in points },
               rotate { Rotation in degrees },
               wt { Weight 0 - 10 },
               italic { Italics flag },
               fixed { Fixed pitch flag },
               0 { All font options },
               wasSet { Set when dlg closes });
  ]
]
```

ForceEvent

Note: Deprecated. Do not use in new code.

Description: Forces the editor to perform an action based on the information provided.

Returns: Nothing

Usage: Script or steady state.

Function Groups: Editor

Related to: AddEditorText | CurrentLine | Editor | GoToOffset | MakeEditor | SetEditMode

Format: ForceEvent(EditorValue, EventType, Parm1, Parm2, Parm3)

Parameters:

EditorValue

Required. An editor value that has been created by MakeEditor.

EventType

EventType	Type of event
-1	Tab
0	Cursor left
1	Cursor right
2	Cursor up
3	Cursor down
4	Enter
5	Delete next character
6	Delete previous character
7	Move cursor to beginning of line
8	Move cursor to end of line
9	Move cursor up one page
10	Move cursor down one page
11	Move cursor to start of the Editor
12	Move cursor to end of the Editor
13	Move the selection block left one character
14	Move the selection block right one character
15	Move the selection block up one line
16	Move the selection block down one line
17	Cut the selection block from the editor to the clipboard
18	Copy the selection block from the editor to the clipboard
19	Insert the text from the clipboard into the editor

Parm1

Required. An expression that has a different meaning based on the event type being forced.

EventType	Parm1 Meaning	Parm1 Value
0 – 24	No meaning	0
25	Offset to which to move the cursor	Byte offset
26	Location to which to move the cursor	X pixel location
27	Text to insert	Text string

Parm2

Required. An expression that has a different meaning based on the event type being forced.

EventType	Parm2 Meaning	Parm2 Value
0 – 24	No meaning	0
25	Highlight characters at new location 1	
??	Don't select any characters 0	
26	Location to which to move the cursor	Y pixel location
27	No meaning	0

Parm3

Required. An expression that has a different meaning based on the event type being forced.

EventType	Parm3 Meaning	Parm3 Value
0 – 24	No meaning	0
25	Characters to high-light	Number of chars
26, 27	No meaning	0

Example:

```
textEd = MakeEditor(){ Create the editor };
...
ForceEvent(textEd { Use a certain editor },
           25 { Move to the given byte offset },
           115 { Move 115 bytes from current location },
           1 { Highlight characters at new location },
           10 { Highlight the next 10 chars });
```

ForceServers

RPC Manager Library

Description: Sets the servership of an application service to a specific state.

Returns: Invalid

Usage: Script Only. (Subroutine call)

Function Groups: Network

Related to:

Format: \RPCManager\ForceServers(Service, ServerStates, [OptGUID])

Parameters:

Service

Name by which the service is known.

ServerStates

A 2D array of servers and their states.

OptGUID

The GUID of the application in which the service instance is located. Optional, the default is the application to which the caller belongs.

Comments: The server states are defined by a 2D array, with row [0] listing the servers and row[1] listing the states.
The servers MUST appear in the same order as returned from GetServersListed for the service. The state values are zero (undetermined), #RPCClient or #RPCServer. These values then get reflected in the RPCStatus to that service.

ForceState

Description: Sets the next state to start when the action script completes.

Returns: Nothing

Usage: Script Only.

Function Groups: Compilation and On-Line Modifications, Logic Control, State

Related to: FirstState

Format: ForceState(State)

Parameters:

State

Required. Any text expression giving the name of the state to start upon completion of the script.

Comments: This statement does not act like an exit point from the script – the script will still run in its entirety. If, however, the script transfer condition did not specify a different state to transfer to, this function will stop the current state. Multiple ForceState statements may be executed in a script, with the last one executed setting the state to which the module will switch.

Example:

```
If TimeOut(1, 1) Main;
[
  IfThen(level < 10, ForceState("Exit"));
]
```

After one second this statement will cause a state transfer to Main. However, if level is less than 10 at that time, the next state will become Exit rather than Main.

FormalParms

Description: Returns the number of formal parameters declared in a module.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Compilation and On-Line Modifications, Advanced Module

Related to: NParm | NumParms | Parameter

Format: FormalParms(Module)

Parameters:

Module

Required. Any expression that returns an object or module type value.

Comments: This function's result may seem obvious, but this function can help automate some of the work in building a parametrized module, just in case the number of parameters declared in a module is changed.

Example:

```
<
TinyModule
(
  a;
  b;
)
[
```

```
  numParms;
]
Main [
  NumParms = FormalParms(Self());
]
>
```

The variable numParms will have a value of 2.

Format

Description: Returns a text string corresponding to numbers in a specified format.

Returns: Text

Usage: Script or steady state.

Function Groups: String and Buffer

Related to: BuffWrite | FWrite | Output | Print | PrintLine

Format: Format(Width, Precision, Value)

Parameters:

Width

Required. Any numeric expression giving the minimum number of characters to use. If fewer characters are required to produce the output, the area is filled with blank spaces on the left to make up the required number of characters. This is useful for aligning numbers on the right.

If more characters are required than the Width parameter specifies, the extra characters are extended to the right. By making Width 0, the output will be aligned on the left.

If the Width parameter is greater than or equal to 100, the format of the floating point number used is in the most compact form which may be in exponential form if the exponent is less than -4 or is greater than the specified Precision parameter.

The actual width used in this mode is 100 less than the

specified Width. Trailing 0s are not displayed in this mode. Values for Width outside the range of 0 to 255 inclusive are invalid.

Precision

Required. Any numeric expression giving the precision of the output (with rounding). It gives the number of digits to appear after the decimal point, if Width is less than 100, or the maximum number of significant digits to appear, if Width is greater than or equal to 100.

Values for Precision outside the range of 0 to 255 inclusive are invalid.

Value

Required. Any numeric parameter giving the number to be formatted.

Comments: This function is useful in conjunction with the Print statement to produce numbers in a printed report. Note that if a text expression is expected as an argument by any VTScada function, and a numeric argument is used, VTScada performs an automatic Format on the value.

Example:

```
number = 123.5692;
textVal = Format(7 { Minimum number of characters },
                 2 { Number of digits after decimal point },
                 number);
```

The variable textVal will be equal to " 123.57"

FormatBatchQuery

(ODBC Manager Library)

Description: When given an array of SQL queries, this module will reformat them into a single query, suitable for a batch call to the specified database.

Returns: Text

Usage: Script Only.

Related to:

Format: \ODBCManager\FormatBatchQuery(dbType, QueryArray, StartingElement, NumberOfElements)

Parameters:

dbType

Required numeric value, indicating the type of this DB connection.

DBType	Meaning
0	MS SQL
1	MS Access
2	Oracle
3	MySQL
4	SyBase

QueryArray

Required. SQL queries to be executed in a batch.

StartingElement

Required. Indicates which element in the array to begin at.

NumberOfElements

Required. Indicates the number of elements of the array that should be processed.

Comments: This module is a member of the ODBCManager Library, and must therefore be prefaced by \ODBCManager\, as shown in "Format" above.
The return value is the batch query as a text string. If the database type is unknown, this function will return only one element from the array, as indicated by the StartingElement. Batch queries cannot be created without knowing which separator string to use.

FormatInteger

Description: Given a numeric value, returns that value converted to Hex, Octal or Binary as specified.

Returns: Text

Usage: Script or steady state.

Function Groups: Math – Generic Functions, String and Buffer

Related to: Format | FormatNumber

Format: \FormatInteger(Value, OutputFormat, Size, LeadingZeros)

Parameters:

Value

Required. Any numeric expression giving the original decimal value to be converted.

OutputFormat

Required. The format to convert to. Constants are used as follows: (Note the leading backslash in each case.)
\#HEX
\#OCTAL
\#BINARY

Size

Required. The size of the result to return. May be one of:
\#DATA_BYTE
\#DATA_WORD
\#DATA_DWORD
\#DATA_QWORD

LeadingZeros

Optional. A Boolean. If TRUE, the output will be padded with leading zeros according to the size specified. Defaults to FALSE.

Comments: Must be preceded by a backslash. Useful for formatting a

numeric value into one of the three available representations.

Example:

```
result = \FormatInteger(25, \#OCTAL, \#DATA_WORD, 0);
```

The variable result will be the "31".

FormatNumber

(System Library)

Description: Given a numeric value, returns a compactly formatted version of this number containing at least the specified number of significant digits.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Math - Rounding Functions

Related to: Format | FormatInteger

Format: \System\FormatNumber(Value, Digits[, IntegerFlag])

Parameters:

Value

Required. Any numeric expression giving the original value to be formatted.

Digits

Required. Any numeric expression giving the number of significant digits to be returned from Value.

IntegerFlag

Optional. Any Boolean expression. If FALSE, integers are returned without modification. Defaults to FALSE.

Comments Must be preceded by \System. This is primarily a significant digits function, but will also return an exponential value if the number of leading or trailing zeros is greater

than three.
If the number is a non-integer, its significant digits are limited (to four digits by default). If the number can be shown with fewer characters using scientific notation, that notation is used. Integers are left untouched unless otherwise directed by an optional parameter.

Example:

```
number = 0.0054321;
result = \System\FormatNumber(number,2);
```

The variable result will be 0.00054

```
number = 0.000054321;
result = \System\FormatNumber(number,2);
```

The variable result will be 5.4e-005

```
number = 0.00455321;
result = \System\FormatNumber(number,2);
```

The variable result will be 0.0046

```
number = 0.00454321;
result = \System\FormatNumber(number,2);
```

The variable result will be 0.0045

```
number = 12345.54321;
result = \System\FormatNumber(number,3);
```

The variable result will be 12300

```
number = 12345;
result = \FormatNumber(number,2);
```

The variable result will be 12345

```
number = 12345;
result = \System\FormatNumber(number,2,1);
```

The variable result will be 12000

FRead

Description: Reads values from a formatted file and returns the number not read.

Returns: Numeric

Usage: Script Only.

Function Groups: File I/O, String and Buffer

Related to: BuffRead | BuffWrite | FileSize | FWrite | SRead | SWrite

Format: FRead([N], File, Offset, Format, V1[, V2, V3, ...])

Parameters:

File

Any text expression specifying the path, file name, and extension to read. A known path alias may be provided in the form, :{KnownPathAlias}.

Offset

Any numeric expression giving the starting file position for the read in bytes, starting at 0.

Format

Required. Any text expression giving the format of how the values (Vn parameters) are to be read.

This format is similar, but not identical, to the C language format string for the scanf function, whereby each of the % format specifications assigns a value to one of the Vn parameters in the statement in the order in which each appears in the list.

Note that like a standard text string, these format specifiers must also be enclosed by double quotes. If a format specification appears for which there are no remaining V parameters, the format specification value is read and discarded.

For the % format specifications, the following form applies (where the [ ] indicates optional elements):
%[*][width]type

Where...
% is mandatory;
The optional asterisk ***** causes the read to occur as per the format specification, but suppresses any assignment to the Vn parameters; and
width is mandatory, specifying the maximum number of characters to read.
The specifications for **type** are listed in the following table:

Note: Note: Format strings are case insensitive. Additionally, specifying a character for a type that is not in this list results in all the characters following the % up to that point to be read exactly as they appear in the Format string and discarded.

Type	Meaning
Nb	Binary format, where n is a number indicating the type of value (see below)
c	Single ASCII character (byte)
d	Signed decimal integer
e	Signed exponential
f	Signed floating point
g	e or f formats
i	Signed decimal integer

l	Line of characters terminated by a carriage return, line feed, or both
n	Present offset in the buffer
o	Unsigned octal
s	Text string
u	Unsigned decimal integer
x	Unsigned hex integer using "abcdef"
znnn	Escape character where nnn is the 3-digit ASCII code

nb, Binary type For the format specification of %nb, where n specifies the type of number, n must be a single digit from one of the following choices. All are low-byte-first.

n value	Type
0	Byte
1	Short integer (2 bytes, low byte first)
2	Long integer (4 bytes, low bytes first)
3	IEEE single precision float (4 bytes)
4	<obsolete>
5	IEEE double precision float (8 bytes)
6	<obsolete>
7	Binary unsigned short (2 bytes, low byte first)
8	Unsigned 32-bit integer

c, ASCII character type: Unlike BuffWrite this type deals with characters in a string; each char-

acter being equal to one byte. Unlike the **%s** option, which reads only up to the first white-space character, the **%c** option reads the number of characters/bytes specified by its width and is not terminated by any particular character. If no width is specified, a single character is read.

d, Signed decimal integer

e, Signed exponential

f, Signed floating point

g, e or f formats

i, Signed decimal integer type: This option normally reads a decimal integer; however, if a leading "0b" is encountered, the number will be interpreted as binary. If a leading "0" (zero only) is encountered, the number will be interpreted as octal. If a leading "0x" is encountered, the number will be interpreted as hexadecimal.

l, Line of characters: This option reads a line of characters terminated by a carriage return, a line feed, or both (in either order). The carriage return and line feed will be discarded, and the next character read will be the first character on the next line. The maximum number of characters read is 4096 (or less if the width option is used).

n, Buffer offset: This option does not read a

value, but returns the present offset in Buffer and can be useful in subsequent reads.

o, Unsigned Octal

s, Text string type: Text in the string is read up until a white-space character is encountered, or the specified width has been read, whichever is smaller. Square brackets enclosing a character, group of characters, or a caret and a group of characters used in the format string reads strings not delimited by spaces. This is a substitute for the %s format specification. The input is read up to the first character that does not appear inside the square brackets (note that this is case-sensitive). A dash may be used to specify a range of characters. For example, the following format specifier:
% [A-Fa-f]
will read a string up to the first which is not an A, B, C, D, E, or F both upper and lower case.

The caret symbol ^. If the first character inside the square brackets is a caret (^), the read progresses up to, but not including, the first character that appears inside the square brackets:
%[^X-Z]
This would read a string up to, but not including, the first X, Y or Z (upper-case only); if the string were terminated by an X, the next character read would be that X. Inside the square brackets, the backslash is used as an escape character - any character following a backslash

(such as a caret, dash, or backslash) is taken as that character without special meaning. For example:
%[^X-Z\^]
would behave as described previously, except that the string would now be read up to but not including the first X, Y, Z, or ^.
Since format specifications for the Vn parameters are indicated by a percentage sign, to read (and discard) an actual percentage sign as part of the text string, precede it with a backslash character (i.e. \%). Also, since the backslash character is used in this manner, as well as with special control characters such as line feed, carriage return and form feed, to read and discard a backslash, use two backslash characters (i.e. \\).

x, Hexadecimal characters: the %x option reads the number of characters/bytes specified by its width and is not terminated by any particular character. If no width is specified, it will continue reading all bytes that can be recognized as hexadecimal characters. For example, given the string "...= 3D", %[^=]=%2x would read the hedadecimal value, 3D (decimal value, 61).

znnn, Escape characters: This specifies an escape character that will be thrown away when read, where nnn is a 3-digit number giving the ASCII character code of the escape character. This option is generally used as the sole format specifier that reads an entire string, spaces

included, discarding every single occurrence of an escape character, or the first occurrence of every pair of escape characters. For example, if the string to be read looked like:

abXc dXXfghiXXXjXXXXkl mX Xn o

and the format specifier indicated that the ASCII code for 'X' (88) was to be the escape code:

%25z088

then the variable that this was read into would contain:

abc dXfghiXjXXkl m n o

Notice that for each occurrence of X, the character immediately following it is saved, even if it is itself an escape character. Then the next occurrence of the escape character is discarded, with the character following it being saved, regardless of what it is, and so on. The width field specifies the maximum number of bytes to place in the output string; if this number is smaller than the input string (less the offending escape characters), the string will be truncated. If no width is specified, a single character will be read.

Control characters: In order to encode certain control characters as part of the Format parameter, one of two methods may be used. The

first is to use a backslash character followed by one of the single character codes listed below to produce the desired result. Please note that the letters must be lower case.

Code	Meaning
\b	Backspace
\f	Form Feed
\n	Line Feed
\r	Carriage Return
\t	Horizontal Tab
\v	Vertical Tab

In addition to the predefined codes, an alternate form may be used:

\nnn: where nnn is a three digit integer in the range of 0 to 255 specifying a certain ASCII character. If the number contains less than three digits, the leading spaces must be padded with zeroes; this is not the case with the previously listed single character control characters. For example, to include the one byte ASCII character G in the output, you could place its decimal equivalent of 71 in the Format string as \071.

V1, V2, V3

Optional. Parameters specifying the variables to be read in the form described by the Format parameter.

Expressions are not allowed.

Each of the Vn parameters is read in the order in which each appears in the parameter list. V1 has

the format given by the first % sequence in the Format parameter, V2 has the second, and so forth.

V1, V2, ...

Required. Are the variables to be read in the form described by the Format parameter. Expressions are not allowed. Each of the Vn parameters is read in the order in which each appears in the parameter list. V1 has the format given by the first % sequence in the Format parameter, V2 has the second, and so on.

Comments In early versions of VTS (WEB), there was a numeric leading parameter, N. This should not be included in any new code.
Data exchange between many file formats is possible if the file formats are known. The return value is optional and is the number of Vn parameters NOT read. This can be used as an error flag.

Example:

```
If ! Valid(err)
[
  err = FRead("G:\RECIPES\GLUE.DAT" { File to read from },
              0 { Starting point },
              "%f%f%n" { Format },
              compoundA, resin, offset { Parameters });
]
```

This reads two ASCII format floating-point numbers, starting at the beginning of G:\RECIPES\GLUE.DAT and places them in compoundA and resin, respectively. After the read is complete, the ending file position is stored in offset. If the read were successful, err is 0. Otherwise it is the number of items which were not read.

Freeze

Description: Freezes all or selected animated graphics in a window.

Returns: Nothing

Usage: Script or steady state.

Function Groups: Graphics, Window

Related to: SelectArea | SelectGraphic | UnselectGraphics

Format: Freeze(Object, Scope, Mode)

Parameters:

Object

Required. Any expression for the object in the window that indicates the selected list to use.

Scope

Required. Any logical expression. If true, all graphics in the window are frozen. If false, only selected graphics in the window are frozen.

Mode

Required. Any numerical expression that controls the freeze, as shown in the following table

Mode	Control Action
0	Stop animation
1	Start animation
2	Toggle animation

Comments: A graphical object that is frozen is no longer updated. It cannot be selected.

Example:

```
Freeze(Self() { Act upon graphics drawn by current module },
       1 { Freeze everything drawn by this module },
       0 { Stop the animation on the frozen objects });
```

FWrite

Description: Writes ASCII or binary data to a file and may also be used

to create or delete a file. It returns the number of data items not written.

Returns: Numeric

Usage: Script Only.

Function Groups: File I/O, String and Buffer

Related to: BuffRead | BuffWrite | FileSize | Format | FRead | Print | PrintLine | Redirect | Save

Format: FWrite([N,] FileSpec, Mode, Position, Format, V1[, V2, V3, ...])

Parameters:

FileSpec

Required. Specifies any output file or printer as the destination to which to write. A known path alias may be provided in the form, :{KnownPathAlias}.
When specifying a printer, FileSpec will accept any of the following:
Local Printer:
Port name (including virtual ports) with or without a trailing colon (e.g. DEF or DEF:. COM1 or COM1:; USB001 or USB001:; etc.)
Windows printer share (e.g. "XYZ Laser Printer")
Windows share name (if the printer is shared) (e.g. "XYZLaser")
Local or Remote Printer:
UNC share name (which includes the host and share name (e.g. "\\localhost\XYlaser" or "\\lab1\NetPrinter")

Mode

Required. Any numeric expression giving the method of opening the file as shown: (further detail follows the table)

Mode	Method of opening
0	Overwrite old data at specified position (other data unaffected)
1	Clear existing data before using file (all data lost)
2	Append data to end of file (other data unaffected)
3	Delete file (all data lost)
4	Open specifically for printer output
5	Print mode, DC-based

If the file does not exist, it will be created if a mode of 0, 1, or 2 is used.

In the case of Mode 2, the Position parameter is ignored. If a position past the end of file is specified, the file will be extended to include the new information. If the file is extended, any information not specified by the Fwrite (such as a gap between the old end of file and the present Fwrite position) will be a string of unknown, random characters (bytes).

In the case of Mode 3, the remaining parameters in Fwrite are ignored, but must be present and valid.

In the case of Mode 4, special handling and error checking for printer output will be provided.

Position

Required. Any numeric expression giving the byte offset (number of characters) from the start of the file, where data is to be written.
This parameter is ignored for Modes 1 - 3 but must be a valid value for all modes. It must be greater than or equal to 0. The start of the file is at Position 0.

Format

Required. Any text expression giving the format of how the values (Vn parameters) are to be written. This format is similar, but not identical, to the C language format string for the printf function, whereby each of the Vn parameters in the statement is assigned to a % format specification in the order in which each appears in the list.
Note that like a standard text string, these format specifiers must also be enclosed by double quotes.
If a format specification appears for which there are no remaining V parameters, the format specification characters themselves are output to the stream exactly as they appear in the Format.
For the % format specifications, the following form applies (where the [ ] indicates optional elements):
%[-][+][SPACE][#][width][.precision]type
where
% (percent sign) is mandatory;
- (minus sign) causes the data to be left justified within the field (for binary types b and ASCII character types c, this option is ignored);
+ (plus sign) causes positive numbers to be prefaced with a + sign (negative numbers are unaf-

fected). This allows easy alignment of positive and negative numbers in a printed column of numbers. For binary types b and non-numerical types, this option is ignored;
space represents the single space character, and is similar to the [+] option but places a single space rather than a plus sign in front of positive numbers (negative numbers are still unaffected). This allows alignment of a column of numbers without having to show all signs. For binary types b and non-numerical types, this option is ignored;
(hash mark) When used with the o , x , or X format, the # flag prefixes any nonzero output value with 0, 0x, or 0X, respectively.
width is a number that specifies the minimum number of characters to output. Numbers that require more characters than specified by the width value are truncated on output. If the number of characters in the number or string is less than width, blanks will be added to the left or right, depending upon whether the output is left or right justified (i.e. whether or not the [-] option has been specified) until the width is reached. For binary types b and ASCII character types c, this option is ignored;
precision has a different meaning for each of the type options as follows:

- Integer types d, l, u, o, x, and X precision specifies the minimum number of digits to output. If the number contains fewer digits, leading zeroes will be added to the left of the number. If precision is 0, omitted, or if the decimal point

appears without a number following it, the precision defaults to 1. The number is not truncated.

- Floating point types e and E precision specifies the number of digits after the decimal point. The last digit is rounded. The default precision in this case is 6. If the precision is 0 or if the decimal point appears without a number following it, no decimal point appears in the output.
- Floating point type f precision specifies the number of digits after the decimal point. The last digit is rounded. The default precision is 0. If the precision is explicitly 0, no decimal point is output. If a decimal point is output, at least one digit will be placed before the decimal point.
- Floating point types g and G precision specifies the maximum number of significant digits to be output. If no precision is specified, all significant digits are written.
- String type s precision specifies the maximum number of characters of the string to be output. If the string contains more characters than specified by the precision, the string is truncated and only the first characters are written. If the precision is not specified, all of the string characters are output.
- ASCII character type c The precision option is ignored.
- Binary type b The precision option is ignored.

x unsigned hex integer using "abcdef"

znnn Escape character where nnn is the 3-digit ASCII code

type is mandatory. The type specification must be one of those listed below.

Note: The case of the letter is important. Specifying a character for the type that is not in this list will result in all the characters following the % up to that point to be output exactly as they appear in the Format string.

Type	Meaning
nb	Binary format, where n is a number indicating the type of value (see below).
c	Single ASCII character (byte)
d	Signed decimal integer
e	Signed exponential; exponent key is "e".
E	Signed exponential; exponent key is "E".
f	Signed floating point.
g	e or f format, whichever is shorter.
G	E or f format, whichever is shorter.
h	Handle to a window.
i	Signed decimal integer.
o	Unsigned octal integer.
p	Pointer to a buffer.
s	Text string.
u	Unsigned decimal integer.
x	Unsigned hex integer using "abcdef".
X	Unsigned hex integer using "ABCDEF".

nb, Binary type For the format specification of %nb, where n specifies the type of number, n must be a single digit from one of the following choices. All are low-byte-first.

n	Value Type
0	Byte, unsigned
1	Signed short integer (2 bytes)
2	Signed long integer (4 bytes)
3	IEEE single precision float (4 bytes)
4	<obsolete>
5	IEEE double precision float (8 bytes)
6	<obsolete>
7	Unsigned short integer (2 bytes)
8	Unsigned long integer (4 bytes)

Note: Other options such as width and precision do not apply to the b type.

c, ASCII character type: This type is not representative of a single character in a string, but rather, represents single byte ASCII characters. Input values (the Vn parameter to which this format specification applies) must be integers in the range of 0 to 255 in order for the output to be a valid ASCII equivalent character. Strings are not acceptable input values. Note that the %c format specifier behaves differently when used in an output statement such as BuffWrite than when used in an input statement, such as BuffRead.

d, Signed decimal integer:

e, Signed exponential: Exponent key is “e”

E, Signed exponential: Exponent key is “E”

f, Signed floating point

g, e or f formats: Whichever is shorter

G, E or F formats: Whichever is shorter

h, Window handle type: This type is used for building structures to be handed to DLLs and should be used by advanced users only.

p, Buffer pointer type: This type is also used for building structures to be handed to DLLs and should be used by advanced users only.

s, Text string type:

Plain text Text in the Format parameter is written exactly as it appears, with three exceptions:

- Percentage sign (%) Since format specifications for the Vn parameters are indicated by a percentage sign, to include an actual percentage sign as part of the Format parameter, precede it with a backslash character (i.e. \%).
- Backslash character (\) Since this is used to indicate special control characters such as line feed, carriage return, and form feed, to write a backslash as part of the Format parameter, use two backslash characters (i.e. \\).
- Quotation marks (") The entire test string is delimited by quotation marks, so to include a set of quotation marks as part of the Format parameter, use a set of quotations marks (i.e. "").

Control characters In order to encode certain control characters as part of the Format parameter, one of two methods may be used. The first is to use a backslash character followed by one of the single character codes listed below to produce the desired result (notice that the letters must be lower case):

Code	Meaning
\b	Backspace

\f	Form feed
\n	Line feed
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab

\nnn In addition to the above predefined codes, \nnn may be used, where nnn is a three digit integer in the range of 0 to 255 specifying a certain ASCII character. If the number contains less than three digits, the leading spaces must be padded with zeroes; this is not the case with the previously listed single character control characters. For example, to include the one byte ASCII character G in the output, you could place its decimal equivalent of 71 in the Format string as \071.

u, Unsigned decimal integer,

x, Unsigned hex integer using "abcdef"

X, Unsigned hex integer using "ABCDEF"

Offset is any numeric expression giving the starting buffer position in characters or bytes for the write, starting at 0.

V1, V2, ...

Required. Are any expressions giving the values to be output in the form described by the Format parameter. Each of the Vn parameters is evaluated and written in the order in which each appears in the parameter list. The way in which they are formatted is dictated by the % format specifications.

V1 is formatted by the first % sequence in the Format

parameter, V2 by the second, and so on. If there are more V parameters than % sequences in the Format string, the remainder are ignored. If there are fewer V parameters than % sequences in the Format string, the remaining % sequences are written literally without any translation.

Comments:

In early versions of VTS (WEB), there was a numeric leading parameter, N. This should not be included in any new code.

You cannot write to a read-only file. You may use GetFileAttribs and SetFileAttribs to get/set the read-only attribute.

If one of the values to be written is outside of the range of the type indicated by the format specifier, a 0 is written. If the value to be written is invalid, nothing is written for most format specifiers, with the exception of %nb, which will write a 0 in the place of the invalid. Invalidity of the output values does not preclude execution of this function.

This function returns the number of Vn parameters not written to the file; a 0 return value indicates success. Variables that contain invalid values that were not written due to their invalidity do not increment this count. An invalid return value indicates an error with one of the parameters. For Mode 3, a 0 indicates success and a 1 (true) indicates a problem deleting the file.

FWrite cannot be used to open COM 1 or COM 2 for direct writing; however, FWrite can connect to a printer on either COM1 or COM2.

All print functions are compatible with the values returned in either of the first two parameters of the PrintDialogBox function.

Example:

```
If timeToWrite;
[
  FWrite("ASCII.DAT" { File to write to },
        1 { Clear data in file },
        0 { Starting point },
        "A=%d\r\nB=%3.2d\r\nC=%6.2f\r\n%8.3s\r\n%c"
        { Format string },
        100.8, 2, 2/3, "finished", 33
        { Values to be written });
  timeToWrite = 0;
]
```

If the variable timeToWrite is true, the FWrite statement would produce a file called ASCII.DAT with five lines of text as follows:

```
A=100
B= 02
C= 0.67
     fin
!
```

G Functions

The sections that follow identify all VTScada functions beginning with "G".

GenerateKey

Description: The GenerateKey function generates a random cryptographic session key or a public/private key pair. A handle to the key or key pair is returned. This handle can then be used as needed with any CryptoAPI function requiring a key handle. It is the VTScada analog of the CryptoAPI's CryptGenKey call.

Returns: Handle

Usage: Script Only.

Function Groups: Cryptography

Related to: DeriveKey | Decrypt | Encrypt | ExportKey | GenerateKey| GetKeyParam | ImportKey | SetKeyParam | Data Encryption and Decryption

Format: GenerateKey(CSPHandle, AlgID [, Flags, Error])

Parameters:

CSPHandle

Required. The handle of a CSP (cryptographic service provider) to use to generate the key.

AlgID

Required. Identifies the algorithm for which the key is to be generated. Values for this parameter vary depending on the CSP used and are defined in WinCrypt.h

Flags

An optional parameter specifying the flags to be passed to CryptGenKey. If omitted or invalid then the value 0 is used.

Error

An optional variable in which the error code for the function is returned. It can have the following values.

Error	Definition
0	Key successfully generated.
1	CSPHandle or AlgID parameters invalid.
x	Any other value is an error from CryptGenKey.

Comments: The return value for this function is a handle to the Key. If an error occurs, then the return value is invalid. A key has a value type of 37. If cast to text, then the hexadecimal value of the algorithm ID will be returned.

Example:

```
[
  Key1;
  Key2;
  Constant CALG_DH_EPHEM = 0xAA02;
  Constant KEY_SIZE = 512;
  Constant CRYPT_EXPORTABLE = 0x00000001;
]
Init [
  If 1 Main;
  [
    { Make a key }
    Key1 = GenerateKey(CSP, CALG_DH_EPHEM,
                       KEY_SIZE << 16 || CRYPT_EXPORTABLE );
  ]
]
```

Get

Note: Deprecated. Do not use in new code.

Description: Reads an array of historical data from a file (written by Save or SaveHistory) and returns the relative file position of the file entry following the last one read, or an error code.

Returns: Array or Error code

Usage: Script Only.

Function Groups: File I/O, Log

Related to: Get | HistorianDeleteRecords | HistorianGetData | HistorianGetInfo | HistorianReadRecords | HistorianWriteRecords | HistorianGetData

Format: Get(ArrayElem, N, File, FieldNum, StartDate, StartTime, TPP, Mode [, StaleTime, PathPrefix])

Parameters:

ArrayElem

Required. Any array element giving the starting point in the array in which data read from the file will be stored. The subscript for the array may be any numeric expression.
If processing a multidimensional array, the usual rules apply to decide which dimension should be used.

N

Required. Any numeric expression giving the number of array entries to use. If this value is greater than the dimension of the array, then the array dimension is used for N.
If N extends past the upper bound of the lowest array dimension, this computation will "wrap-around" and resume at element 0, until N elements have been processed.

File

Required. Any text expression giving the file name for the historical data file. The file extension should not be added to the name since the default of ".DAT" is automatically added.

If the file name is prefixed with a period, the path will be to the directory that the module is contained in.

FieldNum

Required. Any numeric expression giving the field number to be read from the file. The value number for the actual data starts at 0 and corresponds to the columns specified in Save or SaveHistory.

It is also possible to retrieve time data associated with each record by setting this parameter to a negative value.

Time options are:

FieldNum	Time Option
-1	Time of day only
-2	Date only
-3	Time since January 1, 1970

It is possible to retrieve more than one field in a single Get statement. To do this, pass an array of values in as the FieldNum parameter

StartDate

Required. Any numeric expression for the date (expressed as the number of days since January 1, 1970) to search for in the file as the starting date for the data.

If less than 0, this indicates the relative file position to read rather than the date. A -1 indicates the last entry in the file, a -2 the second last, and so on.

If more points are requested than exist, Invalid values

will be retrieved. For example, if this parameter is -20, and there are only 5 data points, the resultant array will contain 15 Invalid values followed by 5 valid values.

Note: If the Save was used with a non-negative Buffers parameter, the last entry in the file will be an Invalid value - the last valid entry will be indicated by -2, the second last by -3 and so on.

StartTime

Required. Any numeric expression giving the time of day (in number of seconds since midnight) on StartDate to search for in the file as the starting time for the data. This may be greater than one day. It may also be negative, where data will start the previous day at (86400 - StartTime) seconds after midnight.

If StartDate is less than 0, StartTime is ignored.

TPP

Required. Any numeric expression giving the time span in seconds for each array entry. Each array element will contain the data that corresponds exactly to this time period, which corresponds to 0 or more data points in the file.

If TPP is positive and FieldNum selects a text value, the first entry which falls in a time is read and Mode is ignored.

If TPP is equal to 0, the data is read from the file and placed in the array on a one-to-one basis. If TPP is less than 0, the data is read on a one-to-one basis from the StartDate and StartTime for (negative) TPP seconds - TPP places an upper limit on the time span to read.

In both of these cases the Mode parameter is ignored.

Mode

Required. Any numeric expression giving the method of handling the data. If TPP is greater than 0, the values that fall in each time span will be represented as follows:

Mode	Representation
0	Time weighted average
1	Minimum in range
2	Maximum in range
3	Change in value over the range
4	Value at start of range
5	Time of minimum in range (in seconds since Jan. 1, 1970)
6	Time of maximum in range (in seconds since Jan. 1, 1970)
7	Count the total number of zero to non-zero transitions within each TPP period.
8	Totalizes, for each TPP, the amount of time when the value is non-zero (Invalid is counted as zero).
9	Totalizes, for each TPP, the arithmetic sum of the recorded values.
10	Interpolates between values.
11	Totalizes, allowing for value rollover.

In the case of modes 5 and 6, FieldNum should still be set to indicate the field number on which the mode is to act. The return values will be times indicating the minimum or maximum in that field for each time span.
In the case of mode 10, the result interpolates

between values. This is primarily intended for the case where there are several TPPs per recorded value, as the time-weighted average returns a stepped curve in this case. The interpolate mode returns a smoother curve.

In the case of mode 11, each value in the time range is compared against the previous value, and if it is less, it is assumed to have rolled over some limit and that limit is then added to the accumulated value. The rollover limit is specified in the StaleTime parameter.

If FieldNum is less than 0 (i.e. a time/date value is being requested), the modes listed above are still valid, although not particularly practical. A time/date will be retrieved for every time span containing a data point, even if that point is Invalid, as in the case of a Save statement whose Buffers parameter is 0 (causing an Invalid value to be written).

If TPP is less than or equal to 0, Mode is ignored. If the data is text, the first entry in a given time range is used for the array entry, and Mode is ignored.

It is possible to retrieve more than one mode in a single Get statement. To do this, pass an array of values in as the Mode parameter.

StaleTime

An optional parameter that sets a maximum validity duration for data elements that are being TPP processed. Normally, every data point is treated as remaining valid until the next data point. If a valid StaleTime parameter is given, then any data point will be treated

as invalid StaleTime seconds after the recorded time. If TPP is less than or equal to 0, StaleTime is ignored. If StaleTime is not required, but PathPrefix is, then StaleTime should be given as either an Invalid value, or a constant zero. If Mode is 11, then StaleTime is the accumulator rollover value.
It is possible to specify more than one stale time in a single Get statement. To do this, pass an array of values in as the StaleTime parameter.

PathPrefix

An optional text expression parameter that enables and controls the retrieval of data from across a set of files.

Comments: This function will return either the relative file position of the file entry following the last entry read, or an error code. If the return value is negative, it is the file position. If it is 0, the end of the file was reached before all the data was read. If the return value is positive, one of the following errors occurred.

Error Code	Error
1	Parameter values out of described range
2	File could not be opened
3	Corrupted .DAT file
4	Field requested could not be found

If StartDate is given a negative value, indicating that a particular entry is to be retrieved, it must be stressed that the file being read by the Get may or may not contain an Invalid record at the end of the file. If the Save that created the file was given a negative number for its Buffers parameter, the Invalid record would not have been written to the file, however, a zero or positive value for Buffers will mean that the last record of the file will be one whose fields are all Invalid and whose time and date stamp reflect the cessation of writ-

ing to the file by the Save.
If FieldNum is an array with more than one element, then Get will retrieve multiple fields from the file. In this case, ArrayElem must represent an array with at least two dimensions. The requested values will be returned in a manner analogous to GetHistory. That is, with the data for a column in the rightmost dimension, and the column index in the previous dimension.
When FieldNum is specified as an array, Mode or StaleTime, or both, may be specified as either a single value or an array of values. If a single value is specified, that value will be used for each of the fields specified in FieldNum. If an array of values is specified, the first element in the array will be applied to the first element of FieldNum, and so on.
If PathPrefix is specified, then this changes the interpretation of the File parameter. In this case, the referenced file is not the source of the data, but a file containing references to other files that are data sources. This file should be in standard VTScada LogFile format and should contain a file reference as the first text value of each record (other values are ignored). The records should be in the correct time order with respect to the data files. The value of the PathPrefix is a string, which when prefixed to one of the file references, will yield a full pathname to the target file. If no prefix is required, but expansion of the dataset is required, then PathPrefix should be an empty string.

- If a filename entry does NOT begin with a "\" or "<drive letter>:\", then the PathPrefix will be prepended to the filename.
- If a filename entry DOES begin with "<drive letter>:\", then the PathPrefix will NOT be prepended to the filename.
- If a filename entry does begin with a "\", then the "<drive letter>:\" from the PathPrefix will be prepended to the filename. If there is no "<drive letter>:\" in the PathPrefix then the "<drive

letter>:\" from the path of the File parameter will be used instead.

PathPrefix would normally be Invalid or the application path.

If a Get is executed while a Save is still active, the save buffer will immediately be flushed to disk. That is to say, data in the buffer that may have been otherwise overlooked by this particular Get will be written to disk so that the current data set is available to the Get in its entirety.

Care must be exercised when using the result from a Get function in further Get functions. This is the normal use for the return value; however, if a Save statement updates the file between Get functions, the return value will be shifted by one from its original file position. The solution to this problem is to either ensure that the Save statement is not executed by disabling the Trigger, or to keep the successive Get functions in a single script.

Example:

```
If ! Valid(exists);
[
  exists = FileFind("G:\Mixer\Trend.DAT"
  { Path and files to search },
  0 { Normal file attributes },
  2 { Return full path and name });
]
If Valid(exists[0]) && exists[0] == "G:\Mixer\Trend.DAT";
[
  Get(Trend[0] { Destination array },
      100 { Get 100 array elements from file },
      "G:\Mixer\Trend" { path and file name },
      0 { Read first 'column' from file },
      Today() { Starting today },
      Seconds() - 3600 { Starting 1 hour ago },
      18 { Each element represents 18 seconds },
      0 { Time weighted avg over 18 secs });
]
```

This tests to make sure that the file exists, and then reads a half hour of data from a file, starting 1 hour (3600 seconds) ago. Note that just after midnight, the expression Seconds( ) – 3600 may be negative. The Get statement interprets this as before midnight on the previous day (which

is correct). X is set to the record following the last record read from the file. Note also that a full path name may be specified, including network drives. Also note that it is irrelevant when data were logged to the file. The Save statement trigger could have been a regular timer (such as AbsTime), or an event (such as Change as applied to a variable).

```
If ! Valid(filePos);
[
  filePos = Get(array1[10]{ Start at element 10 },
                25 { Read 25 array elements },
                "trips" { File trips.dat (in current dir) },
                1 { Read time stamp, not data },
                25 { Read 25 items before EOF },
                0 { Start time is ignored },
                0 { Read on a point-by-point basis },
                0 { Mode is ignored });
]
```

This reads the time stamps of the 25 most recently logged entries into elements 10 through 34 of array1. FilePos is set to the item past the last item read (in this case, the oldest record).

GetAccountID

Security Manager Module

Description: Returns the account ID of the named account.

Returns: String

Usage: Script or steady state.

Related to: GetAccountInfo | GetFullName | GetGroupName | GetUserName | IsLoggedOn | IsSecured | IsSuspended | SecurityCheck | UIErrorToText

Format: \SecurityManager\GetAccountID([AccountName])

Parameters:

AccountName

Optional. The name of the account to obtain the account ID for.

Comments: If AccountName is Invalid or not specified, returns the account ID of the caller's account.

GetAccountInfo

Security Manager Module

Description: Returns one or more AccountData structures.

Returns: Dictionary

Usage: Script Only.

Related to: BuildFullName | GetAccountID | GetFullName | GetGroupName | GetUserName | IsLoggedOn | IsSecured | IsSuspended | SecurityCheck | UIErrorToText

Format: \SecurityManager\GetAccountInfo([AccountID, IndexByID]);

Parameters:

AccountName

Optional. The account ID for which information is to be obtained.

IndexByID

Optional. A Boolean. If TRUE the returned dictionary will use account IDs as the dictionary key. If FALSE (default) the returned dictionary will use the account names as the dictionary key.

Comments: To use this API, the calling code must be running in a security session that has Manager privilege.
If AccountID is valid, a single AccountData structure is returned for the specified account.
If Account ID is Invalid, a dictionary of AccountData structures for every account is returned. The dictionary uses either account names or account IDs as the dictionary key, depending on the value of the IndexByID parameter.
The AccountData structures returned here can be modified in situ and passed back into ModifyAccount or DeleteAccount to effect changes to the account records.
The Password and AltID fields of the returned structure are

always Invalid.

GetAlarmConfiguration

(Alarm Manager module)

Description: Returns a copy of an alarm's configuration structure, returning an unpopulated structure it does not already exist.

Returns: Structure (see comments)

Usage: Script Only.

Function Groups: Alarm

Related to: Commission | GetAlarmStatus

Format: \AlarmManager\GetAlarmConfiguration(AlarmName)

Parameters:

AlarmName

Required text. The alarm name. Typically, the unique id of the alarm tag, or the tag containing built-in alarms.

Comments: GetAlarmConfiguration should be called before commissioning an alarm. This will create an alarm structure that can be populated for the call to \AlarmManager\Commission ().

An alarm has the following configuration structure:

ConfigurationStruct { All Boolean flags default to FALSE }	
Name	Unique name for the alarm
FriendlyName	Display name of the alarm's source
Area	Area
Description	Description. Was "Message" prior to 11.2
Priority	Priority. Must be valid to be commissioned. Must be an integer corresponding to the Alarm Priority tag values.
Reserved	

ConfigurationStruct { All Boolean flags default to FALSE }	
Disable	TRUE to disable the alarm
DisableParmName	Name of the tag's disable parm. Allows us to get the operator name who made the config change.
OnDelay	Seconds to delay before activating
OffDelay	Seconds to delay before clearing
RearmDelay	Seconds to delay before rearming after ack
Setpoint	Setpoint of alarm evaluation
ValueLabels	Array of labels to display instead of Value or Setpoint. Rarely used by tags other than digitals.
Units	Setpoint units
Function	Enumerated function for alarm evaluation (1)
AlarmType	String identifying the type of alarm
Trip	TRUE if alarm only becomes unacked not active
NormalTrip	TRUE if alarm becomes unacked when it clears
OffNormal	TRUE if alarm only becomes active not unacked
Deadband	Setpoint deadband
PopupEnable	TRUE to enable popup display of active alarm
SoundFile	Filename relative to app path of custom sound
Custom	Array/Dictionary/Structure of custom fields
AdHoc	TRUE if alarm is ad hoc

Example:

The following would typically be found in a tag's Refresh state.

```
IfElse(Valid(Name), Execute( { create or obtain the configuration
structure for this alarm }
                            Cfg          = \AlarmMan-
ager\GetAlarmConfiguration(UniqueID);
                              { update the property values in that
structure }
                            Cfg\Name     = Root\UniqueID;
                            Cfg\Area     = AreaValue;
                            Cfg\Priority = PriorityValue;
                            Cfg\Setpoint = 1;
                            Cfg\Function = \AlarmManager\ALM_FUNC_
```

```
EQUAL;
                                    { commission (or update the commission
of) the alarm }
                                 \AlarmManager\Commission(Root, Cfg,
Value);
                              );
);
```

Related Information:

[1]Function constants are documented in the VTScada Programmer's Guide in: "Alarm Manager Function Constants"

GetAlarmList

(Alarm Manager module)

Description: Returns filtered and sorted lists of records from alarm databases.

Returns: See descriptions in the parameter list.

Usage: Script Only.

Function Groups: Alarm

Related to:

Format: \AlarmManager\GetAlarmList(ListPtr, LengthPtr, ListNames[, Snapshot, IncludeShelved, IncludeConfig, FilterExpr, CalculatedFields, CalculatedFieldsScope, SortOrder, BeginTime, EndTime, MaxHistory, PtrDone, PtrHistoryProgress, PtrChangeStats, DBTags, RefreshInterval, Realm])

Parameters:

ListPtr

Required. This pointer will be assigned the array of records being returned. The array may be longer than the number of records, see LengthPtr for the valid length.

LengthPtr

Required. Pointer to a variable which will be assigned

the number of elements in the ListPtr array that are to be used.

ListNames

Required. Provide an array of alarm names to which the alarm lists will subscribe, or a text string containing a single alarm name. Valid list names include:

- History
- Active
- Unacked
- Current
- Shelved
- Disabled
- Configured

Snapshot

Optional Boolean. If TRUE, then this function will take a snapshot of the list(s) and terminate. If FALSE, then the function will build the list while also keeping it updated.
A snapshot of live list(s) acts as a subroutine, all other uses must launch a worker process.
Defaults to FALSE.

IncludeShelved

Optional Boolean. Set TRUE to include shelved alarms. Defaults to FALSE.

IncludeConfig

Optional Boolean. Set TRUE to include alarm configuration events in History. Defaults to FALSE.

FilterExpr

Optional text string that is the expression to evaluate if a record is to be included.
Standard area filtering is handled automatically so

should not be included in this expression.

CalculatedFields

Optional array. May be used to create additional record fields or modify existing records. A call-back subroutine allows record field values to be replaced, for example swapping AccountID for an operator name.

If specified, this parameter must be an array that defines the extra record fields.

For example:

```
CalculatedFields = New(4);
CalculatedFields[0] = "FriendlyName";
CalculatedFields[1] = "IsActive";
CalculatedFields[2] = "IsUnacked";
CalculatedFields[3] = "IsDisabled";
```

CalculatedFieldsScope

Optional. Scope in which to find calculated field values. Often, Self()

SortOrder

Optional. For live lists this is sorting information for SortArray.

For example:

```
{ Sort by friendly name }
SortInfo = SortKeys("FriendlyName", 1 { text
}, #SortAscending);
```

For history set TRUE for forward chronological, and FALSE for reverse (default).

BeginTime

Optional UTC timestamp. Oldest time. May be used when filtering history.

EndTime

Optional UTC timestamp. Newest time. May be used when filtering history.

MaxHistory

Optional numeric. Limits the number of history records retrieved. No default.

PtrDone

Optional pointer. Will be set TRUE when history search completes.

PtrHistoryProgress

Optional pointer. Will be set to a structure instance containing progress stats (HistoryProgressDef)

PtrChangeStats

Optional pointer. Will be set to a structure with Add/Mod/Del counts. Allows the caller to detect changes.

DBTags

Optional array of AlarmDatabase tag names (UniqueIDs or Friendly Names). If a simple text value, then it is the single tag to access. If Invalid then all AlarmDatabase tags are included. Realm filtering is automatically detected and used.

RefreshInterval

Optional. Minimum time in seconds between sorted array updates. Defaults to a half-second.
May also be set by adding an application property, AlarmListRefreshDelayTime.

Realm

Optional; realm to use for realm area filtering. This will be detected automatically for the logged-on account if not otherwise specified.

Comments: An alarm database contains a history of alarm transactions / event records. Event records are included only in a queries for a history list.
Filtering - Only records that pass filtering will be

displayed. Filtering may be:

- Area filtering (Realms, etc..) [automatic]
- Shelved alarm suppression
- Custom filter expression

For efficiency reasons the filtering is done on the base record rather than the calculated version.
History - Unlike the live lists, the history is unbounded and must be accessed from storage. The process can take time so search progress is reported via PtrHistoryProgress.
When working with a live list, the system will automatically rerun filtering and recalculate fields if an alarm is changes on any list; allowing filtering and calculated fields based on other list state.

Example:

```
LoadList [
  If 1 WaitResults;
  [
    \AlarmManager\GetAlarmList(&Records, &NRecords, "History", TRUE,
ShowShelved, FALSE,
                               Concat("Record\Name == """,
Record\Name, """ && (Record\Action == ""Active"" || Record\Action ==
""Trip"")"),
                               Invalid, Invalid, FALSE, StartTime,
EndTime, NRecordsLimit,
                               &Done, &Progress);
  ]
]
```

GetAlarmObject

(Alarm Manager module)

Description: Returns an alarm object value given an alarm name.

Returns: Object reference.

Usage: Script Only.

Function Groups: Alarm

Related to:

Format: \AlarmManager\GetAlarmObject(AlarmName[, ptrRoot])

Parameters:

AlarmName

Required text. The alarm name. Typically, the unique id of the alarm tag, or the tag containing built-in alarms.

ptrRoot

Optional pointer. If valid, and if the alarm is a tag or a submodule of a tag, that tag's object value will be returned in this parameter.

Comments: This function can also provide the alarm's Root tag value, returned through the optional second parameter.

Example:

```
{ Given an array of alarm records, get the tag object from the alarm
name whenever the array index, SelRecord, changes. }
If Watch(1, Records[SelRecord]\GUID);
[
  \AlarmManager\GetAlarmObject(Records[SelRecord]\Name, &Root);
  ShortName = \AlarmManager\GetNameOfRecord(Records[SelRecord]);
]
```

GetAlarmStateStats

(Alarm Manager module)

Description: Returns a structure containing the cumulative alarm state statistics for the specified tag. If the tag is the ancestor of multiple alarm tags, the stats will be the accumulation of all descendent alarms.

Returns: Structure

Usage: Script Only.

Function Groups: Alarm

Related to: GetAlarmObject

Format: \AlarmManager\GetAlarmStateStats(TagName)

Parameters:

TagName

Required. The tag for which statistics are to be gathered.

Comments: TagName may be a context or other parent, in which case statistics will be gathered for all of the child tags.

Example:

```
TotalStats = Code\AlarmManager\GetAlarmStateStats(MyContextTagName);
TotalUnackedPriority = TotalStats\HighestUnackedPriority;
```

GetAlarmStatus

(Alarm Manager module)

Description: Returns a reference to an alarm's status structure, providing access the alarm's current state without having to make function calls.

Returns: Structure (see comments)

Usage: Script Only.

Function Groups: Alarm

Related to: IsActive | IsDisabled | IsShelved | IsUnacked

Format: \AlarmManager\GetAlarmStatus(AlarmName)

Parameters:

AlarmName

Required text. The alarm name. Typically, the unique id of the alarm tag, or the tag containing built-in alarms.

Com- Use this when writing code that will monitor the status of an

ments: alarm. In older versions of VTScada, function calls were used for the same task. For the sake of backward-compatibility, those functions will continue to work, but should not be used in new code.

AlarmStatus Struct	
IsActive	TRUE if alarm is on the Active list };
IsUnacked	TRUE if alarm is on the Unacked list };
IsShelved	TRUE if alarm is on the Shelved list };
IsDisabled	TRUE if alarm is on the Disabled list };

Variables within the structure may be accessed using either the dot notation or the backslash.

Example:
Monitor for when the alarm is active and unacknowledged, possibly to display a message.

```
  IF Watch(1) Main;
  [
     AlmStatus = \AlarmManager\GetAlarmStatus(AlarmName);
  ]
...
Main [
  IfElse(AlmStatus.IsActive && AlmStatus.IsUnacked,
       ... { TRUE case  },
       ... { FALSE case });
]
```

Related Information:

GetAppInstance

(System Library)

Description: Retrieves the Layer object (LayerRoot) for a particular application specified by GUID.

Returns: Object (Layer object returned via a parameter)

Usage: Script Only.

Function Groups: Configuration Management

Related to: GetLoadedAppInstance | GetCodeObj | GetGUID

Format: \System\GetAppInstance(GUID, pAppInstance)

Parameters:

GUID

Required text. The 36-byte GUID of the application to be retrieved.

pAppInstance

A pointer to a variable. The Layer object matching the GUID will be returned via this pointer.

Comments: The return value of this function will be an object that becomes invalid upon completion. If the Layer has not finished loading, this function will wait until it does so before returning. This is an asynchronous operation.
If the GUID does not match any known application, then an Invalid value will be assigned to the second parameter.

Examples:

```
Init [
  IF 1 WorkState;
  [
    WaitObj = \System\GetAppIntance(GUID, &TargetLayer);
  ]
]
```

GetByte

Description: Returns a single byte from a buffer.

Returns: Byte

Usage: Script or steady state.

Function Groups: String and Buffer

Related to: SetByte

Format: GetByte(Buffer, Offset)

Parameters:

Buffer

Required. Any buffer expression giving the buffer to get the byte from.

Offset

Required. Any numeric expression giving the offset from the start of the buffer in bytes, starting from 0.

Comments: This function is useful for manipulating ASCII text on a byte-by-byte level. For example, examining serial I/O driver response packets.

Example:

```
oneByte = GetByte(response, 0);
```

The variable oneByte is set to the ASCII value of the first byte in the text variable response. If response is Invalid, then oneByte will be Invalid. If oneByte is valid, its value will always be in the range 0 to 255.

GetClientDiverts

(RPC Manager Library)

Description: Returns a one-dimensional array of flags, indicating the divert status of each client.

Returns: Array

Usage: Steady State only.

Function Groups: Network

Related to: SetDivert | GetClientGUIDs

Format: \RPCManager\GetClientDiverts(Service [, OptGUID])

Parameters:

Service

Required. Any text expression giving the name by which the service is known.

OptGUID

An optional parameter that is any expression giving the 16-byte binary form of the globally unique identifier (GUID) for the application in which the service instance is located. The default is the application to which the caller belongs.

Comments: This module is a member of the RPC Manager's Library, and must therefore be prefaced by \RPCManager\, as shown in the "Format" section. If the application you are developing is a script application, the module call must be prefaced by System\RPCManager\, and the System variable must be declared in AppRoot.src.
Each client that has service RPCs diverted to an auxiliary holding queue (as a result of the SetDivert function) has its array element in the returned array set to one.
If the 16-byte binary format of the GUID for the application in which the service instance is located is not known, the GetClientGUIDs function may be used to obtain it.

GetClientGUIDs

(RPC Manager Library)

Description: Returns a one-dimensional array of the application GUIDs of the clients of the specified RPC service instance.

Returns: Array

Usage: Steady State only.

Function Groups: Network

Related to: GetClientIPs | GetClientList

Format: \RPCManager\GetClientGUIDs(Service [, OptGUID]);

Parameters:

Service

Required. The name by which the service is known.

OptGUID

An optional parameter that is any expression giving the 16-byte binary form of the globally unique identifier (GUID) for the application in which the service instance is located. The default is the application to which the caller belongs.

Comments: This subroutine is a member of the RPC Manager's Library, and must therefore be prefaced by \RPCManager\, as shown in the "Format" section. If the application you are developing is a script application, the subroutine call must be prefaced by System\RPCManager\, and the System variable must be declared in AppRoot.src.
The returned array elements will all be the same, unless cross-application RPC is being used.

GetClientIPs

(RPC Manager Library)

Description: Returns a one-dimensional array of the IPs of the clients of the specified service instance.

Returns: Array

Usage: Steady State only.

Function Groups: Network

Related to: GetClientGUIDs | GetClientList | GetClientNodes

Format: \RPCManager\GetClientIPs(Service [, OptGUID]);

Parameters:

Service

Required. The name by which the service is known.

OptGUID

An optional parameter that is any expression giving the 16-byte binary form of the globally unique identifier (GUID) for the application in which the service

instance is located. The default is the application to which the caller belongs.

Comments: This subroutine is a member of the RPC Manager's Library, and must therefore be prefaced by \RPCManager\, as shown in the "Format" section. If the application you are developing is a script application, the subroutine call must be prefaced by System\RPCManager\, and the System variable must be declared in AppRoot.src.
On a multi-homed system, any of the possible IPs may be returned for each client.

GetClientList

(RPC Manager Library)

Description: Returns a one-dimensional array of the names of the clients of the specified service instance.

Returns: Array

Usage: Steady State only.

Function Groups: Network

Related to: GetClientGUIDs | GetClientIPs

Format: \RPCManager\GetClientList(Service [, OptGUID]);

Parameters:

Service

Required. The name by which the service is known.

OptGUID

An optional parameter that is any expression giving the 16-byte binary form of the globally unique identifier (GUID) for the application in which the service instance is located. The default is the application to which the caller belongs.

Comments: This subroutine is a member of the RPC Manager's Library, and must therefore be prefaced by \RPCManager\, as

shown in the "Format" section. If the application you are developing is a script application, the subroutine call must be prefaced by System\RPCManager\, and the System variable must be declared in AppRoot.src.
On a multi-homed system, any of the possible alias names may be returned for each client.

GetClientMode

(RPC Manager Library)

Description: Returns a one-dimensional array of the modes of the clients of the specified service instance.

Returns: Array

Usage: Steady State only.

Function Groups: Network

Related to:

Format: \RPCManager\GetClientMode(Service [, OptGUID]);

Parameters:

Service

Required. The name by which the service is known.

OptGUI

An optional parameter that is any expression giving the 16-byte binary form of the globally unique identifier (GUID) for the application in which the service instance is located. The default is the application to which the caller belongs.

Comments: This subroutine is a member of the RPC Manager's Library, and must therefore be prefaced by \RPCManager\, as shown in the "Format" section. If the application you are developing is a script application, the subroutine call must be prefaced by System\RPCManager\, and the System variable must be declared in AppRoot.src.

The mode value represents the current synchronization state of the client, with respect to the local service instance. Presently, one of the following values may be returned in each array element
\RPC_ACCEPT_ALL – the client is fully synchronized;
\RPC_SYNC_MODE – the client is being synchronized; or
\RPC_LINKCONTROL_ONLY – client requires synchronization.

GetClientNodes

(RPC Manager Library)

Description: Returns a one-dimensional array of the object values of the Machine Nodes of the clients of the specified service instance.

Returns: Array

Usage: Steady State only.

Function Groups: Network

Related to:

Format: \RPCManager\GetClientNodes(Service [, OptGUID]);

Parameters:

Service

Required. The name by which the service is known.

OptGUID

An optional parameter that is any expression giving the 16-byte binary form of the globally unique identifier (GUID) for the application in which the service instance is located. The default is the application to which the caller belongs.

Comments: This subroutine is a member of the RPC Manager's Library, and must therefore be prefaced by \RPCManager\, as shown in the "Format" section. If the application you are

developing is a script application, the subroutine call must be prefaced by System\RPCManager\, and the System variable must be declared in AppRoot.src.

GetCodeObj

Description: Retrieves the "Code" object associated with the layer.

Returns: Object

Usage: Script Only.

Function Groups: Configuration Management

Related to: GetLoadedAppInstance |

Format: LayerRoot\GetCodeObj()

Parameters: none

Comments: This function is useful when there is a need to work with the code object of a layer other than the current application.

Examples:

Given a valid, 32-character application GUID, stored in TextGUID, the following will obtain the list of parameters in the current page.

```
Layer = GetLoadedAppInstance(TextGUID);
AppCodeObj = Layer\GetCodeObj();
PageParmNames = ListVars(Scope(AppCodeObj, SessionData), "*", 0, 0, 4
{parms}, 0, 0, 0, 0);
```

GetColorInfo

Description: Returns the brush and pen information for a given graphic statement.

Returns: Numeric

Usage: Script Only.

Function Groups: Graphics, Color

Related to: Brush | Pen

Format: GetColorInfo(Code, PenBrushNum, Attribute)

Parameters:

Code

Required. Any statement which gives the code pointer of the object whose color information is desired.

PenBrushNum

Required. Any numeric expression which gives the number of the pen or brush which the color information is desired for. The first pen or brush is numbered 1.

Attribute

Required. Any numeric expression which gives the desired color attribute.

Attribute	Color Attribute
0	Type of the PenBrushNum supplied. Return value is 19 (brush) or 20 (pen)
1	The foreground color of the brush or pen specified
2	For a brush, the background color, for a pen, the width
3	For a brush, the pattern number, for a pen, the style

Comments: The return value of this function is determined by Attribute.

GetConfiguration

Description: Returns the configuration parameters from the license key for this copy of VTScada.

Returns: Numeric

Usage:	Script or steady state.
Function Groups:	Software and Hardware
Related to:	SerialNum
Format:	GetConfiguration(Option)
Parameters:	
	Option

Option	Configuration Item
0	Serial number of this copy of VTS
1	Number of hours for restricted run time (0 if unrestricted)
2	Expiry date for free updates'
3	Number of tags permitted (or 0 if no limit)
4	Number of browser clients permitted
5	Run mode (0 = Full. 1 = Run Time. 2 = Configuration. 3 = View Mode)
6	Dongle required to run.
7	Returns the number of days remaining in the evaluation period. If this is not an evaluation license, then this option returns Invalid.
8	Returns the Client Connection Restriction. If TRUE (1). VTScada client systems installed with this license may only connect to servers with the same serial number. If FALSE (0), VTScada client systems installed with this license may connect to any server.
9	The VTScada Alarm Notification System is an option controlled by the installation key in VTScada versions 7.1 and later. Option 9 returns the status of this enable – TRUE if the Alarm Notification System is supported. FALSE if the Alarm Notification System is not supported.
10	Returns the copyright statement built into VTScada. The format of the

Comments: If the license key information fails certain consistency checks, then this function returns Invalid for all values of Option.
This function can be used to determine license restrictions pertaining to the current copy of VTScada. GetConfiguration(0) is identical to the function SerialNum() and is preferred for all new code.

GetConnList

(ODBC Manager Library)

Description: Called to obtain a list of the available connections. The list will be returned in the form of two pointers; one for an array of the connection objects and the other for an array of the connection names.

Returns: Numeric. The connection list will be passed back in the parameters.

Usage: Script or steady state.

Related to:

Format: \ODBCManager\GetConnList(ObjPtr, NamePtr)

Parameters:

ObjPtr

Required. The list of object values to return.

NamePtr

Required. The list of names to return.

Comments: This module is a member of the ODBCManager Library, and must therefore be prefaced by \ODBCManager\, as shown in "Format" above.
Returns 0 upon completion
The module may be called as a subroutine in a script or as a called function in a state. When called in a state, there is no automatic updating of the connection list as it changes.

GetContainerNumActive

(Alarm Manager module)

Description: Returns the number of active alarms within a hierarchy of tags.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Alarm Manager

Related to: Accumulate | GetContainerNumUnacked

Format: GetContainerNumUnacked(ContainerObj);

Parameters:

ContainerObj

Required. The container tag to query.

Comments: The alarm count is accumulated by the hierarchical accumulator module. Alarm tags and tags with built in alarms are designed to contribute to this accumulator automatically. If you have written a custom tag with a built-in alarm, you must ensure that it contributes its active alarms to the count.

Example:

```
TotalNumUnacked = Code\AlarmManager\GetContainerNumActive(MyTagName);
```

GetContainerNumUnacked

(Alarm Manager module)

Description: Returns the number of unacknowledged alarms within a hierarchy of tags.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Alarm Manager

Related to: Accumulate | GetContainerNumActive

Format: GetContainerNumUnacked(ContainerObj);

Parameters:

ContainerObj

Required. The container tag to query.

Comments: The alarm count is accumulated by the hierarchical accumulator module. Alarm tags and tags with built in alarms are designed to contribute to this accumulator automatically. If you have written a custom tag with a built-in alarm, you must ensure that it contributes its unacknowledged alarms to the count.

Example:

```
TotalNumUnacked = Code\AlarmManager\GetContainerNumUnacked
(MyTagName);
```

GetContributors

Description: Returns a copy of an array of object values of contributors for a given container.

Returns: Array

Usage: Script Only.

Function Groups: Containers and Contributors

Related to:

Format: GetContributors(HandleName, ContainerObj);

Parameters:

HandleName

Required. The name of the handle variable in the container module.

ContainerObj

Required. The object VTScada Value Types - Numeric

Reference of the container tag module.

Comments: This function can be called from the contributor.

GetCryptoProvider

Description: The GetCryptoProvider function is used to acquire a handle to a particular key container within a particular cryptographic service provider (CSP). This returned handle can then be used to make calls to the selected CSP. It is the VTScada analog of the CryptoAPI CryptAcquireContext call.

Returns: Handle

Usage: Script Only.

Function Groups: Cryptography

Related to: DeriveKey | Decrypt | Encrypt | ExportKey | GenerateKey | GetKeyParam | ImportKey | SetKeyParam

Format: GetCryptoProvider(CSPType [, CSPName, ContainerName, Flags, Error])

Parameters:

CSPType

Required. The type of CSP required. Values are defined in WinCrypt.h.

CSPName

An optional parameter that holds the name of the required CSP. If omitted or invalid, then a handle to the default CSP of the specified type will be acquired.

ContainerName

An optional parameter that holds the name of the key container. If omitted or invalid, then the default key container for the CSP is used.

Flags

An optional parameter specifying the flags to be passed to CryptAcquireContext. If omitted or invalid then the value 0 is used.

Error

An optional variable in which the error code for the function is returned. It has the following meaning

Error	Meaning
0	CSP handle successfully returned.
1	CSPType parameter invalid.
x	Any other value is an error from CryptAcquireContext.

Comments: The return value for this function is a handle to the CSP. If an error occurs, then the return value is invalid. A CSP handle has a value type of 36. If cast to text then the name of the CSP will be returned.
If ContainerName is omitted or invalid then a default key container name is used. For example, the Microsoft Base Cryptographic Provider uses the logon name of the user logged on as the key container name. Other CSPs can also have default key containers that can be acquired in this way.

Example:

```
[
  CSP;
  CSPFail;
  Container = "VTS";
  Constant PROV_DSS_DH = 13;
  Constant CRYPT_NEWKEYSET = 8;
  Constant NTE_BAD_KEYSET = 0x80090016;
]
Init [
  If 1 Main;
  [
    CSP = GetCryptoProvider(PROV_DSS_DH, Invalid,
                            Container, Invalid, CSPFail);
    IfThen(CSPFail == NTE_BAD_KEYSET,
     { Not used this container before, make a new one }
    CSP = GetCryptoProvider(PROV_DSS_DH, Invalid,
```

```
                          Container, CRYPT_NEWKEYSET, CSPFail);
    );
  ]
]
```

GetDefaultValue

Description: Returns a variable's default value.

Returns: Varies

Usage: Script or steady state.

Function Groups: Compilation and On-Line Modifications, Variable

Related to: SetDefault | FindVariable

Format: GetDefaultValue(Variable)

Parameters:

Variable

Required. Any expression for the variable value.

Comments: If the given variable does not have a default value this function will return Invalid. The current value of Variable does not affect the return value of this function.

Example:

```
If ! Valid(defValue);
[
  defValue = GetDefaultValue(FindVariable("originalVal", Self(),
                                          0, 1));
]
```

The above statement will assign the default value of original to defValue. The reason that the statement is inside a script is because FindVariable may only appear inside a script.

GetDevices

(VoiceTalk Module)

Description: Runs in the VoiceTalk thread and returns a list of devices available on a SAPI text-to-speech stream.

Returns: Array

Usage: Script Only.

Function Groups: Speech and Sound

Related to: Configure | GetVoices | Reset | ShowLexicon | Speak | VoiceTalk

Format: VoiceTalkStream\GetDevices()

Parameters:

VoiceTalkStream

Required. A speech stream returned from VoiceTalk.

Comments: This function will immediately return a 1-dimensional list of output device names available for the text-to-speech stream. The strings in this array are suitable to pass as devices to the VoiceTalk\Configure module.

Example:

```
sHandle = \VoiceTalk();
If Valid(sHandle) && ! getDevices;
[
  getDevices = 1;
  sDevices = sHandle\GetDevices();
]
```

This will return an array of all available text-to-speech output devices in the array sDevices.

GetFileAttribs

Description: Returns information about the specified file.

Returns: Numeric

Usage: Script only.
May be used in optimized Tag Parameter Expressions.

Function Groups: File I/O

Related to: SetFileAttribs

Format: GetFileAttribs(FileName[, Mode])

Parameters:

FileName

Required. Any text expression giving the name of the file. A known path Known Path Aliases for File-Related Functions may be provided in the form, : {KnownPathAlias}.

Mode

Optional numeric value that controls what information is returned by this function. Defaults to 0 if missing or invalid.

Mode:	Function Returns:
0	File attributes in the form of a value set to the sum of the following values:
1	Timestamp showing the date modified
2	File access and status flags as follows:

Value	Bit No.	Attribute
0	-	Normal
1	0	Read only
2	1	Hidden
4	2	System
8	3	Archive
16	4	Directory

- 0 File is locked or doesn't exist.
- 1 Open access (file exists and isn't open-locked.
- 2 Read access (bit 0 is always true if this is true).
- 3 Write access (bit 0 is always true if this is true).

Comments: The return value will vary according to Mode. See tables above.

Example:

```
If Watch(0, newFile);
[
```

```
  attribs = GetFileAttribs(Concat(MyPath, newFile));
]
```

The above statement will cause attribs to be set to the file attribute value of newFile every time its name changes.

GetFullName

Security Manager Module

Description	Returns the full, namespace-qualified name of the caller's account
Returns	String
Usage	Script or steady state.
Related to:	BuildFullName | GetAccountID | GetAccountInfo | GetGroupName | GetUserName | IsLoggedOn | IsSecured | IsSuspended | SecurityCheck | UIErrorToText | See also, "Security NameSpaces" in the VTScada Programmer's Guide.
Format	\SecurityManager\GetFullName()
Parameters	None
Comments	None

GetGroupName

Security Manager Module

Description	Returns the namespace of the caller's account.
Returns	String
Usage	Script or steady state.
Related to:	BuildFullName | GetAccountID | GetAccountInfo | GetFullName | GetUserName | IsLoggedOn | IsSecured | IsSuspended | SecurityCheck | UIErrorToText | See also, "Security NameSpaces" in the VTScada Programmer's Guide.

Format \SecurityManager\GetGroupName()

Parameters None

Comments None

GetGUID

Description: Creates a globally unique identifier or converts an existing GUID to another format.

Returns: Text

Usage: Script only.
May be used in optimized Tag Parameter Expressions.

Function Groups: Network

Related to:

Format: GetGUID(Format [, ExistingGUID])

Parameters:

Format

Required. Any numeric expression giving that defines the format of the return value as follows

Format	Format definition
0	36 byte ASCII string
1	16 byte binary string

ExistingGUID

An optional parameter that is any text expression for the GUID to be converted from one format to another.
No default value.

Comments: Any generated GUID will be different from all others generated on this or any other machine now or in the future. It is particularly useful in generating unique variable or file names.
The current application's GUID is stored in

\LocalGUID, using the 16-byte binary form.

Example:

The following script will cause GUID1 to be set to a globally unique 16 bit binary string and GUID2 to convert GUID1 to its 36 byte ASCII string equivalent.

```
If 1 Main;
[
  GUID1 = GetGUID(1);
  GUID2 = GetGUID(0, GUID1);
]
```

The next example obtains the current application's existing GUID as a 32-character string.

```
If 1 Main;
[
  MyGUID = GetGUID(0,\LocalGUID);
]
```

GetHistory

Note: Deprecated. Do not use in new code.

Description:	Get History from a File written by Save or SaveHistory. This threaded function retrieves an array of data from a .DAT file for a certain time span. If the parameters to GetHistory are valid and an attempt is made to get the data, the return value is 0, otherwise, if no attempt is made to get the data, the return value is 1.
Returns:	Numeric (see discussion of the first parameter)
Usage:	Script Only.
Function Groups:	File I/O, Log
Related to:	GetHistory | HistorianDeleteRecords | HistorianGetData | HistorianGetInfo | HistorianReadRecords | HistorianWriteRecords | Get | GetLog| GetLogInfo | Save | SaveHistory | TGet
Threaded:	Yes

Format: GetHistory(Array, File, StartTime, EndTime [, Error, PathPrefix])

Parameters:

Array

Required. A variable that will be set to an array upon completion of the data retrieval. The format of the array is [column][record]. Column 0 is the timestamp in seconds since January 1, 1970. Each subsequent column is the data for that record. Array is not set if there is no data for the requested period.

File

Required. Any text expression giving the file name for the historical data file. The file extension must be included. If the file name is prefixed with a period, the path will be to the directory in which the module is contained.

StartTime

Required. Any numeric expression giving the start time of the period for which data is requested in seconds since January 1, 1970.

EndTime

Required. Any numeric expression giving the end time of the period for which data is requested in seconds since January 1, 1970.

Error

An optional variable that will always be set to a valid value upon completion of the GetHistory. Its meaning is as follows:

Error	Error Description
0	No error
1	Parameter values out of described range
2	File could not be opened
3	Corrupted .DAT file
4	Field requested could not be found

If Error is not required, but PathPrefix is, then Error should be given as an Invalid value.

PathPrefix

An optional text expression parameter that enables and controls the retrieval of data from across a set of files.

Comments:

This function is threaded - that is to say, it starts its own thread and VTScada will continue executing. When it is finished executing, it will set the data in Array. Note that Array will not be initially invalidated upon execution of this statement, so if Array already contained data when the GetHistory was executed, that data will remain untouched until all of the data requested by GetHistory has been amassed, at which time Array will be set to its new value.

Since Array could conceivably remain invalid indefinitely (i.e. there was no data in the requested time span), Error will always be set to a valid value to indicate completion of execution.

There is a return value for this function that indicates if any of its parameters are invalid. The function will immediately return a value of false (0) unless a parameter was invalid, in which case it will return true (1). Note that the return value only signals completion of the function's execution if it is true, otherwise the function will continue executing in its thread.

If PathPrefix is specified, then this changes the interpretation of the File parameter. In this case, the referenced file is not the source of the data, but a file containing references to other files which are the data sources. This file should be in standard VTScada log file format and should contain a file reference as the first text value of each record (other values are ignored). The records should be in the correct time order with respect to the data files. The value of the PathPrefix is a string, which when prefixed to one of the file references, will yield a full pathname to the target file. If no prefix is required, but expansion of the dataset is required, then PathPrefix should be an empty string.

- If a filename entry does NOT begin with a "\" or "<drive letter>:\", then the PathPrefix will be prepended to the filename.
- If a filename entry DOES begin with "<drive letter>:\", then the PathPrefix will NOT be prepended to the filename.
- If a filename entry does begin with a "\", then the "<drive letter>:" from the PathPrefix will be prepended to the filename. If there is no "<drive let-

ter>:" in the PathPrefix then the "<drive letter>:" from the path of the File parameter will be used instead.

PathPrefix would normally be Invalid or the application path.

GetHostByName

Description: Calls the WinSock "gethostbyname" function and returns the host name, address(es) and alias names for the named computer.

Returns: Array

Usage: Script only.
May be used in optimized Tag Parameter Expressions.

Function Groups: Network

Related to: WkStaInfo

Format: GetHostByName(Name)

Parameters:

Name

Required. Any text expression giving the host name (or an alias) for which information is required.

Comments: This function will return details of the TCP/IP interface to the named computer. The return value is an array with three elements comprising a structure as follows:

Element Information

Official host name of the requested computer as known to the system. If using DNS or a similar resolution system, it is the Fully Qualified Domain Name (FQDN) that caused the server to return a reply. If using a local "hosts" file, it is the first entry

after the IP address.
An array of addresses. Each address is a text string giving a "dotted quad" address (xx.xx.xx.xx). If the name being queried is a remote machine, then there will be only one address. If the name being queried is the local machine, then there will be one address for each network interface (including RAS), and no interpretation should be placed on the ordering of the returned addresses.
Invalid, or an array of alias names. These are alternative names, defined in the local hosts file, by which the target computer may be referenced.
If the name passed to this function is irresolvable as belonging to a computer on the network, the reply will be invalid.
Depending upon the network topology, it may be several seconds before this call returns, although other threads in the application will continue to run.

GetID

Description: This returns the ID (opcode) of a given function.

Warning: For use by advanced programmers only. Effective use of this function requires a thorough understanding of VTScada programming.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Compilation and On-Line Modifications

Related to: Compile

Format: GetID(Statement)

Parameters:

Statement

Required. Any expression for the code value of the function.

Comments: This function is used by the compiler. The OpCode for all functions can be found in the file WEBFUNC.TXT

GetInhibitedServiceList

RPC Manager Service

Description: Returns a one-dimensional array of the names of all services inhibited from RPCManager servership control.

Returns: Array

Usage: Script Only.

Function Groups: Network

Related to:

Format: \RPCManager\GetInhibitedServiceList([OptGUID])

Parameters:

OptGUID

The GUID of the application in which the service instance is located. Optional, the default is the application to which the caller belongs.

Comments: Requires that the application be running.

GetINIProperty

(System Library)

Description: Given an array of INIProperty structures, returns the value of a given property from that array.

Returns: Value

Usage: Script Only.

Function Groups: Configuration Management, Variable

Related to: CaptureSettings | ReadPropertiesFile | SetINIProperty | WritePropertiesFile

Format: \System\GetINIProperty(InputArray, Name[, Comment, pFail])

Parameters:

InputArray

Required. An array of INIProperty structures. See Comment section.

Name

Required. The name of the property whose value is to be returned.

Comment

Optional pointer to a text value. The comment associated with the property will be returned in this field.

pFail

Optional Boolean pointer. If the property is not found, FALSE will be returned to the calling module in this parameter.

Comments: The INIProperty structure is as follows

```
INIProperty Struct [
    Name        { Variable name in the .star-
tup/.dynamic file  };
    Value       { Simple value
};
    Comment     { Text comment if present in the
file          };
    Hidden      { TRUE if not visible in Edit
Properties GUI   };
  ];
```

Example:

```
  { Read Settings.Startup file }
TempProperties = ReadPropertiesFile(Concat(AppPath,
                                           #APP_INI_FILENAME,
                                          #STATIC_INI_EXT));
GUID    = GetINIProperty(TempProperties\Sections["Application"],
```

```
"GUID");
OEMGUID = GetINIProperty(TempProperties\Sections["Application"],
"OEMGUID");
```

GetInSyncServers

Description: Returns a one-dimensional array of the names or IPs of the potential, synchronized servers for the given service.

Returns: Array

Usage: Steady State only.

Function Groups: Network

Related to:

Format: \RPCManager\GetInSyncServers(Service, [OptGUID])

Parameters:

Service

Name by which the service is known.

OptGUID

The GUID of the application in which the service instance is located. Optional, the default is the application to which the caller belongs.

Comments: None

GetInstance

Description: Returns the object value of a module instance.

Warning: For use by advanced programmers only. Effective use of this function requires a thorough understanding of VTScada programming.

Returns: Object

Usage: Script Only.

Function Groups: Compilation and On-Line Modifications, Basic Module

Related to: FindVariable | CalledInstances | NumInstances

Format: GetInstance(Module, Index)

Parameters:

Module

Required. Any expression for the code value of the module.

Index

Required. Any numeric expression indicating which instance of Module to get. The most recently started instance is instance 0.

Comments: If the number requested is larger than the total number of instances for the module, the return value will be invalid.

GetIP

(RPC Manager Library)

Description: Returns an IP address for a workstation, given its name.

Returns: Text

Usage: Steady State only.

Function Groups: Network

Related to:

Format: \RPCManager\GetIP(Name)

Parameters:

Name

Required. Any of the names by which the remote workstation is known to the RPC Manager.

Comments: This subroutine is a member of the RPC Manager's Library, and must therefore be prefaced by \RPCManager\, as shown in the "Format" section. If the application you are developing is a script application, the subroutine call must be prefaced by System\RPCManager\, and the System variable must be declared in AppRoot.src.

GetKeyCount

Description: Return a count of the number of keys stored by the given dictionary.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Dictionary

Related to: ListKeys | GetNextKey

Format: GetKeyCount (Dictionary)

Parameters:

Dictionary

Required. Any dictionary for which you wish to retrieve the key count.

Comments: Generally used in combination with ListKeys. The single parameter is required and must contain a dictionary, otherwise INVALID will be returned.

GetKeyParam

Description: The CryptGetKeyParam function retrieves data that governs the operations of a key. It is the VTScada analog of the CryptoAPI's CryptGetKeyParam call.

Returns: Varies

Usage: Script Only.

Function Groups: Cryptography

Related to: DeriveKey | Decrypt | Encrypt | ExportKey | GenerateKey | GetCryptoProvider | ImportKey | SetKeyParam

Format: GetKeyParam(Key, Param [, Flags, Error])

Parameters:

Key

Required. The handle to the key being queried.

Param

Required. A parameter specifying the query being made. Values are defined in WinCrypt.h

Flags

An optional parameter specifying the flags to be passed to CryptGetKeyParam. If omitted or invalid then the value 0 is used.

Error

An optional variable in which the error code for the function is returned. It may have the following values

Error	Meaning
0	Key parameter successfully returned.
1	Key or Param parameters invalid.
x	Any other value is an error from CryptGetKeyParam.

Comments: The parameter for the key is returned. If an error occurs, the return value is invalid.
The allowable values for Param vary with the key type.

Example:

```
[
  KeyG;
  Constant KP_G = 12 { DSS/Diffie-Hellman G value };
]
Init [
If 1 Main;
  [
    { Get the key parameter }
    KeyG = GetKeyParam(Key1, KP_G);
  ]
]
```

GetLoadedAppInstance

Description: Retrieves the Layer object (LayerRoot) for a particular application specified by GUID

Returns: Layer object

Usage: Script Only.

Function Groups: Configuration Management

Related to: GetAppInstance | GetCodeObj

Format: GetLoadedAppInstance(GUID)

Parameters:

GUID

Required text. The 36-byte GUID of the application to be found.

Comments: This function is similar to GetAppInstance, but is a synchronous operation, returning the Layer object directly rather than in a pointer parameter.
If the target application is not yet loaded, then this function will return Invalid rather than wait.

GetLocalIP

(RPC Manager Library)

Description: Returns an IP address for the local workstation that is known to the specified remote workstation.

Returns: Text

Usage: Steady State only.

Function Groups: Network

Related to:

Format: \RPCManager\GetLocalIP(Name);

Parameters:

Name

Required. Any of the names by which the remote workstation is known to the RPC Manager.

Comments: This subroutine is a member of the RPC Manager's Library, and must therefore be prefaced by \RPCManager\, as shown in the "Format" section. If the application you are developing is a script application, the subroutine call must be prefaced by System\RPCManager\, and the System variable must be declared in AppRoot.src.

GetLocalNumber

(RPC Manager Library)

Description: Returns the index of the local workstation down the prioritized server list for the named service.

Returns: Numeric

Usage: Steady State only.

Function Groups: Network

Related to:

Format: \RPCManager\GetLocalNumber(Service [, OptGUID]);

Parameters:

Service

Required. The name by which the service is known.

OptGUID

An optional parameter that is any expression giving the 16-byte binary form of the globally unique identifier (GUID) for the application in which the service instance is located. The default is the application to which the caller belongs.

Comments: This subroutine is a member of the RPC Manager's Library, and must therefore be prefaced by \RPCManager\, as

shown in the "Format" section. If the application you are developing is a script application, the subroutine call must be prefaced by System\RPCManager\, and the System variable must be declared in AppRoot.src.
The machine at the top of the prioritized server list will return an index of zero. If the local machine cannot be a server for the specified service, -1 is returned.

GetLog

Description: This launched module returns an array of logged data. GetTagHistory should be considered for use in new code.

Returns: An array, stored in the first parameter (see comments)

Usage: Script Only.

Function Groups: File I/O, Log

Related to: GetTagHistory

Format: \GetLog(&Result, Tag, Name, Start, End, TPP, Size, Mode, Obsolete [, StaleTime] [, TimeZoneBias])

Parameters:

Result

Required. A pointer to set to the array retrieved from file. The array may be multidimensional as described in the Comments section. Please review the information carefully.

Tag

Required. The object value of the tag from which to get the logged data.

Name

Required. The name of the logged value in the tag. If Invalid or a valid empty string, a timestamp is used.
It is possible to retrieve more than one field in a single GetLog statement. To do this, pass an array of values

in as the Name parameter.

Start

Required. The start time in seconds since January 1, 1970.

End

Required. The end time in seconds since January 1, 1970. If invalid, Size limits the number of tags retrieved.

TPP

Required. The time span per tag. If Invalid, Size limits the number of tags retrieved.

Size

Required. The maximum number of tags to get. If greater than 0, data is accessed one at a time from file, rather than at equal time intervals. If TPP is valid, Size is ignored.

Mode

Required. Indicates the mode of data collection.
Note: The mode is useful only when the TPP parameter is valid and greater than 0. Mode may be one of:

Mode	Data Collection
0	Time-weighted average
1	Minimum in range
2	Maximum in range
3	Change in value over the range
4	Value at start of range
5	TimeOfMinInRange
6	TimeOfMaxInRange
7	SumOfZtoNZTransitions
8	SumOfNonZeroTime
9	Totalizer
10	Interpolated
11	Difference between the start and end values of a range (see comments)

It is possible to retrieve more than one mode in a single GetLog statement. To do this, pass an array of values in as the Mode parameter.

Obsolete n/a

No longer used – maintained for backward compatibility.

StaleTime

The meaning of StaleTime depends on the selected Mode.
For Mode 0 through 10, StaleTime is indicates the maximum time to propagate an old value.
For Mode 11, this is the maximum value of a range

(see comments).

It is possible to specify more than one stale time in a single GetLog statement. To do this, pass an array of values in as the StaleTime parameter.

TimeZoneBias

Used with time zone aware reports. If included, it should be the number of seconds that your time zone differs from UTC, specified in time stamps. This is the value that would be returned from TimeZone(0).

Comments:

When the value of the Result parameter becomes valid, all of the data has been retrieved.

If Name is an array with more than one element, then GetLog will retrieve multiple fields from the tag. In this case, Result will be a two-dimensional array. The requested values will be returned in a manner analogous to GetHistory. That is, with the data for a column in the rightmost dimension, and the column index in the previous dimension.

When Name is specified as an array, Mode or StaleTime, or both, may be specified as either a single value or an array of values. If a single value is specified, that value will be used for each of the fields specified in Name. If an array of values is specified, the first element in the array will be applied to the first element of Name, and so on.

When TimeZoneBias is included, GetLog will translate the start and end times for the requested range of data from the local time zone to that of the server before retrieving the data. Then, before returning the retrieved data, it will translate the timestamps back to the local time zone.

When Mode is set to 11, Getlog counts to a maximum value as set by StaleTime, and then rolls over to start a new interval. Whenever there is a decrease in value from one record to the next in the file, it is assumed that the maximum of the range has been reached and a rollover has occurred.

Upon rollover, the value of Getlog for the previous interval is calculated as (StaleTime + CurrentValue) – Minimum Value from previous interval. Thus, for a StaleTime set to 100 and values read as follows: 10, 40, 90, 5.

The transition from 90 to 5 marks a rollover event. The value for the previous interval is (100 + 5) – 10 = 95.

This module executes in parallel to the launching code. Watch for the return array to be filled with valid data to determine when it has finished.

Example:

```
  If 1 Loop;
  [
     { Setup the TimeZone Bias }
    TZBias  = \IsTimeZoneAware ? TimeZone(0) : Invalid;

     { Set up the fieldnames, modes & staletime arrays for Getlog }
    FieldNames     = New(5);
    FieldNames[0]  = Invalid { Timestamps };
    FieldNames[1]  = "Quality";
    FieldNames[2]  = "SuccessCount";
    FieldNames[3]  = "ErrorValue";
    FieldNames[4]  = "ErrorAddress";

    Modes          = New(5);
    Modes[0]       =
    Modes[1]       =
    Modes[2]       =
    Modes[3]       =
    Modes[4]       = Invalid;

     { Retrieve Data for the first tag }
    \GetLog(&TagData,
            Scope(\Code, Tags[TagIndex]) { Point object value      },
            FieldNames                   { Read data               },
            Start                        { Start time              },
            End                          { End time                },
            Invalid                      { Time per point          },
            #NUMRECLIMIT                 { No max number of recs   },
            Modes                        { Modes                   },
            Invalid                      { Obsolete                },
            Invalid                      { Stale Time              },
            TZBias                       { TimeZone Bias
});
```

GetLogInfo

Note: Deprecated. Do not use in new code.

Description: Interrogates a historical data file, or a set of historical data

files, and returns overall time, date, and record count information either for the entire file(set), or for a specified time range.

Returns: Nothing (data returned in parameters)

Usage: Script Only.

Function Groups: File I/O, Log

Related to: GetLogInfo | HistorianDeleteRecords | HistorianGetData | HistorianGetInfo | HistorianReadRecords | HistorianWriteRecords | Get | GetHistory | Save | SaveHistory | TGet

Format: GetLogInfo(File, Earliest, Latest, NRecords [, PathPrefix, StartTime, EndTime]);

Parameters:

File

Required. The historical data file (or index file for a file set) for which information is required.

Earliest

Required. Any variable in which will be returned the earliest time stamp from the file(set).

Latest

Required. Any variable in which will be returned the latest time stamp from the file(set).

NRecords

Required. Any variable in which will be returned the total number of records in the file(set).

PathPrefix

An optional text expression parameter that enables and controls the retrieval of data from across a set of files.

StartTime

An optional timestamp parameter that defines the start time of the range to be examined.

EndTime

An optional timestamp parameter which defines the end time of the range to be examined.

Comments:

For any optional parameter that is to be set, all optional parameters preceding the desired one must be present, although they may be Invalid.
All timestamps are specified in seconds since January 1, 1970.
If the number of records found is zero, then Earliest and Latest are both set Invalid.
If PathPrefix is specified, then this changes the interpretation of the File parameter. In this case, the referenced file is not the source of the data, but a file containing references to other files which are the data sources. This file should be in standard VTScada logfile format and should contain a file reference as the first text value of each record (other values are ignored). The records should be in the correct time order with respect to the data files. The value of the PathPrefix is a string, which when prefixed to one of the file references, will yield a full pathname to the target file. If no prefix is required, but expansion of the dataset is required, then PathPrefix should be an empty string.

- If a filename entry does NOT begin with a "\" or "<drive letter>:\", then the PathPrefix will be prepended to the filename.
- If a filename entry DOES begin with "<drive letter>:\", then the PathPrefix will NOT be prepended to the filename.

- If a filename entry does begin with a "\", then the "<drive letter>:" from the PathPrefix will be prepended to the filename. If there is no "<drive letter>:" in the PathPrefix then the "<drive letter>:" from the path of the File parameter will be used instead.

PathPrefix would normally be Invalid or the application path.

GetMachineNode

(RPC Manager Library)

Description: Returns the object value of the MachineNode for the specified name or IP.

Returns: Object

Usage: Steady State only.

Function Groups: Network

Related to:

Format: \RPCManager\GetMachineNode(Name);

Parameters:

Name

Required. Any of the names or IPs by which the workstation is known to the RPC Manager.

Comments: This subroutine is a member of the RPC Manager's Library, and must therefore be prefaced by \RPCManager\, as shown in the "Format" section. If the application you are developing is a script application, the subroutine call must be prefaced by System\RPCManager\, and the System variable must be declared in AppRoot.src.

GetMakeAltPtr

(RPC Manager Library)

Description: Returns a pointer to a variable containing the Alternate status for the local service instance in the calling application for the specified service. Steady state or subroutine call.

Returns: Pointer

Usage: Steady State only.

Function Groups: Network

Related to:

Format: \RPCManager\GetMakeAltPtr(Service);

Parameters:

Service

Required. The name by which the service is known.

Comments: This subroutine is a member of the RPC Manager's Library, and must therefore be prefaced by \RPCManager\, as shown in the "Format" section. If the application you are developing is a script application, the subroutine call must be prefaced by System\RPCManager\, and the System variable must be declared in AppRoot.src.

GetModuleRefBox

Description: Returns the outer reference box for any selectable (GUI) graphics in a module.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Compilation and On-Line Modifications, Advanced Module, Graphics

Related to: GetXformRefBox | SetModuleRefBox

Format: GetModuleRefBox(Module, Option)

Parameters:

Module

Required. Any expression for the code value of the module.

Option

Required. Any expression that defines the return value as indicated by the following:

Option	Return Value
0	left side
1	bottom side
2	right side
3	top side
4	width
5	height

Comments:

A module's reference box defines an area that will exactly enclose all layered (GUI) graphics in the module regardless of state before any rotations and trajectories have been applied. A module reference box, or MRB as it is sometimes called, is not a clipping region and objects can and often will extend outside of their MRB as a result of applied rotations or trajectories.

When a module is transformed, the transform is based on the size of the module as determined by its reference box. The module's reference box will always exactly fill the reference box of the transform. In the case of graphics that have had a rotation or trajectory applied to them, the graphics will be transformed correctly, but the MRB may no longer contain the objects in their modified positions.

If a SetModuleRefBox has been done within the module in question, or its module reference box has been defined by following the module name with the four boundaries enclosed in parentheses, the return value will be based on

these explicitly defined boundaries. Otherwise the return value will define the minimum reference box that will exactly enclose all (GUI) graphics in the module. This does not include any child graphics (i.e. graphics created by child modules of Module).
Module does not need to be running at the time that this statement is called in order to retrieve valid data.

Example:

```
If ! Valid(rightSide);
[
  rightSide = GetModuleRefBox(Self(), 2);
]
```

Variable rightSide will contain the X-coordinate for the graphic object drawn by this module that is the furthest right in the window.

GetModuleText

Description: Returns information about a module's document file.

Warning: For use by advanced programmers only. Effective use of this function requires a thorough understanding of VTScada programming.

Returns: Varies

Usage: Script or steady state.

Function Groups: Compilation and On-Line Modifications, Advanced Module

Related to: AdjustCode | GetOneParmText | GetParmText | GetStateText | GetTransitText | GetVariableText | SetModuleText | SetOneParmText | SetParmText | SetStateText | SetTransitText | SetVariableText | TextOffset | TextSize

Format: GetModuleText(Module, Info)

Parameters:

Module

Required. Any expression for the module.

Info

Required. Any numeric expression giving the information to return, as shown in the following table:

Info	Information to return
0	File name which defines Module
1	Character offset to beginning of Module
2	Length of Module
3	Character offset to beginning of parameter definitions
4	Character offset to beginning of variable definitions
5	Character offset to beginning of state definitions
6	Character offset to beginning of child module definitions
7	Character offset to first variable definition
8	Length of variable definitions
9	Character offset to first parameter definition
10	Length of parameter definitions

Comments: This function is used when automatically modifying modules.

GetNameOfRecord

(Alarm Manager module)

Description: Given an alarm record, returns the tag name.

Returns: Text

Usage: Script Only.

Function Groups: Alarm

Related to: GetAlarmObject

Format: \AlarmManager\GetNameOfRecord(AlarmRecord[, ParentTag])

Parameters:

AlarmRecord

Required record. A reference to the alarm, as returned by the function GetAlarmObject.

ParentTag

Optional. The parent tag's name. If valid, this value will be stripped from the result.

Comments: This function is intended primarily for presentation purposes.
The function will return whichever of the following, in order, for which it can find a valid value: the relative tag name, the full name, or the unique ID (alarm name).

Example:

```
{ Given an array of alarm records, get the tag object from the alarm
name whenever the array index, SelRecord, changes. }
If Watch(1, Records[SelRecord]\GUID);
[
  \AlarmManager\GetAlarmObject(Records[SelRecord]\Name, &Root);
  ShortName = \AlarmManager\GetNameOfRecord(Records[SelRecord]);
]
```

GetNextKey

Description: Allows a linear search through a dictionary in place. i.e. without copying the contents to an array.

Returns: Varies

Usage: Script Only.

Function Groups: Dictionary

Related to: ListKeys

Format: GetNextKey(Dictionary[, Key, Order, KeyFound])

Parameters:

Dictionary

Required. Any dictionary you wish to search.

Key

The key to start from. If this is invalid, or if the key is not found in the dictionary, then the first key in the given order will be returned.

Order

An optional numeric expression. Defines the search according to the following table of values. Defaults to 0 if missing. Setting this parameter to Invalid will result in the function returning Invalid.

Order	Meaning
0	Forward alphabetic search
1	Forward ordinal search. Returns the next newer key. Begins at the oldest key if the parameter Key is invalid.
2	Backward alphabetic search
3	Backward ordinal search. Returns the next older key. Begins at the newest key if the parameter Key is invalid.

Key Found

Upon completion of the function, this value will receive the key of the record found, which matches with the

value of that record returned by the function. If there are no further keys in the given order, this value will be set to INVALID.

Comments: The return value is the value associated with the key found, or INVALID if no key was found.
Only the first parameter is required.

GetNumUnacked

(Alarm Manager module)

Description: Returns either the number of unacknowledged alarms or a Yes/No flag to indicate that the alarms exist. A filter may be applied to limit the search.
Note that this function will place a heavy load on computer resources while searching.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Alarm

Related to: Register (Alarm Manager) | DBListGet

Format: \AlarmManager\GetNumUnacked(UserFilter, ReturnYesNoFlag);

Parameters:

UserFilter

Required. An array that can be used to limit the search. If set to Invalid, GetNumUnacked will return all unacknowledged alarms.
This will typically be a one-dimensional, three-element array with elements as follows:

Element	Description
0	Index of the field in the alarm table. This is defined as the second parameter to the KeyName as shown in the KeyName table
1	Limiting value
2	Comparison Operator (see Comparison Operator table)

KeyName Table

Value to use for filter (*)	Alarm Table Key Names
0	Message
1	Priority
2	Type
3	HookPointValue
4	Area
5	HookPointUnits
6	Operator

(*) Note: these values may vary if an application has specified a different table header and indexes in their own customized Settings.Startup.
Comparison Operator Table

Comparison Value	Comparison	Case Sensitive
0	Equal to	no
1	Greater than	no
2	Less than	no

ReturnYesNoFlag

Required. A Boolean that controls the result returned by the function.
If true, only a 1 or a 0 will be returned, indicating whether unacknowledged alarms were found.
If false, the number of unacknowledged alarms will be returned.

Example:

```
Temp = New(3);
Temp[0] = \AlarmPriorityField;
Temp[1] = 1; {Critical }
Temp[2] = 0; { Equal to, not case sensitive. This parameter may hold
a 1-dimensional array with 2 or 3 numeric elements. }
\AlarmManager\GetNumUnacked(Temp,  0);
```

If a more detailed filtering criterion is required, a 2-dimensional array may be used. See DBListGet for examples and further information.

GetOEMLayer

Description: Retrieves the layer root module of the OEM layer (should one exist) of the layer this is called against.

Returns: A pointer (within the parameter)

Usage: Script Only.

Function Groups: Configuration Management

Related to: GetWCPath |

Format: \Layer\GetOEMLayer

Parameters:

OEMLayerPtr

Required. A variable, into which a pointer to the OEM layer's root module will be placed.

Comments: The return value of the function will be set to Invalid upon completion. A pointer to the OEM layer's module will be returned in the first parameter to this function.

Examples:

```
MyLayer\GetOEMLayer(ParentLayer);
```

GetOneParmText

Description: Returns the text for one parameter of a function.

Warning: For use by advanced programmers only. Effective use of this function requires a thorough understanding of VTScada programming.

Returns: Text

Usage: Script Only.

Function Groups: Compilation and On-Line Modifications, Advanced Module

Related to: AdjustCode | GetModuleText | GetParmText | GetStateText | GetTransitText | GetVariableText | SetModuleText | SetOneParmText | SetParmText | SetStateText | SetTransitText | SetVariableText | TextOffset | TextSize

Format: GetOneParmText(Function, Parameter[, Type])

Parameters:

Function

Required. Any expression for the code value of the function.

Parameter

Required. Any numeric expression indicating which parameter's text to return.

Type

Optional numeric expression. Controls what will be returned by the function according to the following table:

Type	Description
0	default. Return the text for the parameter.
1	Return the offset in the file for the start of the parameter.
2	Return the size (number of bytes) of the parameter.

GetOutputTypes

Description: Returns a list of available report output type plugins.

Returns: Array

Usage: Script Only.

Function Groups: Report

Related to: GetReportTypes

Format: \ReportPanel\GetOutputTypes(PtrTypeNames, PtrTypeMods)

Parameters:

PtrTypeNames

Required. A pointer to a variable storing output names.

PtrTypeMods

Required. A pointer to a variable storing output module names.

Comments: This subroutine, declared in ReportPanel, is a plug-in to help with custom retrieval of output types. GetOutputTypes must be prefaced by \ReportPanel\, as shown in the "Format" section above.

GetOverrides

Description Returns an array of OpCodes and the module value that will run when each OpCode is executed

Warning For use by advanced programmers only. Effective use of this function requires a thorough understanding of VTScada programming.

Returns 2-dimensional array

Usage Script Only.

Function Groups Compilation and On-Line Modifications, Advanced Module

Related to: SetOverride

Format GetOverrides(TargetModule)

Parameters

TargetModule

Required. Any expression that can be resolved to a module value.

Comments The target module will be queried for a list of overridden OpCodes. The return value is a 2 dimensional array where the first dimension is the override as set by SetOverride and second dimension will be [0] for the OpCode and [1] for the corresponding module that has been set to override that opcode.
Together with SetOverride, this provides the ability to override a built-in function in a module with a module call. It is used primarily for testing. See SetOverride for full details.

GetParameter

Description: Returns the requested parameters as a constant, variable

or code pointer.

Warning: For use by advanced programmers only. Effective use of this function requires a thorough understanding of VTScada programming.

Returns: Constant, variable or code value.

Usage: Script Only.

Function Groups: Advanced Module

Related to:

Format: GetParameter(Code, Index)

Parameters:

Code

Required. Any expression for the code value or code pointer of the function.

Index

Required. Parameter number to obtain. Starts with 0 for the first parameter.

Comments: If the parameter being retrieved is a constant number, then GetParameter just returns that number. The same goes for a constant string parameter.
If the parameter is just a variable, then GetParameter returns the variable (a value of type \#VTypeVariable).
If the parameter is itself a function, then GetParameter returns a code value for it.

GetParmText

Description: Returns the text for all parameters of a function.

Warning: For use by advanced programmers only. Effective use of this function requires a thorough understanding of VTScada programming.

Returns: Text array

Usage: Script Only.

Function Groups: Compilation and On-Line Modifications, Advanced Module

Related to: AdjustCode | GetModuleText | GetOneParmText | GetStateText | GetTransitText | GetVariableText | SetModuleText | SetOneParmText| SetParmText | SetStateText | SetTransitText | SetVariableText | TextOffset | TextSize

Format: GetParmText(Function)

Parameters:

Function

Required. Any expression for the code value of the function.

Comments: This function may only appear in a script.

GetParserOffset

Description: Returns the offset before the last compiled statement.

Warning: For use by advanced programmers only. Effective use of this function requires a thorough understanding of VTScada programming.

Returns: Numeric

Usage: Script Only.

Function Groups: Compilation and On-Line Modifications

Related to: Compile

Format: GetParserOffset(ParserStack)

Parameters:

ParserStack

Required. Any expression for the parser stack value.

Comments: This is used by the compiler to give the location of an

error.

GetPathBound

Description: Returns the bounding box coordinates for a path.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Graphics

Related to: GetModuleRefBox | Path

Format: GetPathBound(Path, Object, Transform, Side)

Parameters:

Path

Required. Any expression that returns a Path value.

Object

Required. Any object expression which defines the window where Path is drawn.

Transform

Required. Any expression for the transform value applied to Path.

Side

Required. Any numeric expression for the coordinate to return:

Side	Side to Return
0	Left
1	Bottom
2	Right
3	Top

GetPlatformInfo

Description: Gathers information about the current application and the workstation it is running on.

Returns: Numeric status indicator. (Platform information is returned in the parameter)

Usage: Script Only.

Function Groups: Configuration Management

Related to: Platform | WKStaInfo

Format: Layer\GetPlatformInfo(&Info)

Parameters:

&Info

Required. Pointer to a variable. The information gathered will be returned in a structure that this variable points to.

Comments: This utility function makes use of WkStaInfo and Platform to gather relevant information about the application. The structure returned has the following format:

```
INIFiles Struct [
  FileName { file name and extension for the
file };
  OEM { TRUE if an OEM layer file };
  Workstation { Name of the workstation or
invalid if global };
  Layer { Instance of application layer owning
the file };
  Dynamic { TRUE if a dynamic property };
  Sections { Dictionary of sections, each ele-
ment of which
is an array of Property structures (see fol-
lowing) };
  Changed { User sets to true if the file has
been changed,
initialized to false };
]
```

The property structures have the following format:

```
INIProperty Struct [
  Name { Variable name in the settings file };
  Value { Simple value or an ordered array of
values if the
variable occurs more than once in the section of
the file };
  Comment { Text comment if present in the file
};
  Hidden { TRUE if not shown in the Edit Prop-
erties GUI };
]
```

Examples:

GetPowerState

Description: Returns a structure that holds details of the computer's power supply status.

Returns: Structure containing four elements

Usage: Steady State only. See: Rules for Usage.

Function Groups: Network & Workstation

Related to:

Format: GetPowerState()

Parameters: none

Comments: This function is intended for use when the workstation is running on battery power, whether UPS or laptop battery. The return structure contains the following elements:
PowerStatus (Short. AC line status. 0 == battery, 1 == A/C, 2 == backup power.)
BatteryState (Short. Indicates any of High, Low, Critical, Charging or No Battery.)
BatteryLevel (Short. The remaining charge in the battery, expressed as a percentage.)
BatteryLifetime (Double. Estimated number of

seconds of charge remaining in battery.)

Examples:

```
{ Display selected page name on the title bar }
PowerState = GetPowerState();
```

GetReferencedValues

Description: Collects all dynamically referenced values in the call tree rooted at the parameter and returns them in an array. Returns Invalid if none are found.

Returns: Array

Usage: Script Only.

Function Groups: Variable Functions

Related to:

Format: GetReferencedValues(Object)

Parameters:

Object

Required. The object for which referenced values are to be found.

Comments: Referenced objects are found via quaffles in steady state. (Quaffle[1])

Examples:

```
{ Get an array of pointers to all values scoped-to in the call tree
rooted at PageObj. }
ReferencedVals = GetReferencedValues(PageObj);
```

GetRemoteVersion

(RPC Manager Library)

---

[1] A hidden compiler function that forms a dynamic relationship between a value and a statement. Essential to the operation of Steady State code.

Description: Returns the version number of VTScada running on a specified workstation. Steady state or subroutine call.

Returns: Numeric

Usage: Steady State only.

Function Groups: Network

Related to:

Format: \RPCManager\GetRemoteVersion(NameOrIP);

Parameters:

NameOrIP

Required. Any of the names or IPs by which the workstation is known to the RPC Manager.

Comments: This subroutine is a member of the RPC Manager's Library, and must therefore be prefaced by \RPCManager\, as shown in the "Format" section. If the application you are developing is a script application, the subroutine call must be prefaced by System\RPCManager\, and the System variable must be declared in AppRoot.src.

GetReportTypes

Description: This subroutine returns a list of available report type plugins.

Returns: Array

Usage: Script Only.

Function Groups: Report

Related to: GetOutputTypes

Format: \ReportPanel\GetReportTypes(PtrTypeNames, PtrTypeMods)

Parameters:

PtrTypeNames

Required. A pointer to a variable storing report names.

PtrTypeMods

Required. A pointer to a variable storing report module names.

Comments: This subroutine, declared in ReportPanel, is a plugin to help with custom retrieval of report types. GetReportTypes must be prefaced by \ReportPanel\, as shown in the "Format" section above.

GetReturnValue

Description: Returns a module's return value.

Returns: Varies

Usage: Script or steady state.

Function Groups: Compilation and On-Line Modifications, Basic Module

Related to: Return | ResetParm

Format: GetReturnValue(Object)

Parameters:

Object

Required. An object expression for the module instance whose return value is required.

Comments: This function is useful for obtaining a return value from a module that has been called as a parameter to another module, and whose return value has been or will be altered by a ResetParm statement.

GetSelected

Description: Returns a selected graphic item in a window.

Returns: Object

Usage: Script Only.

Function Groups: Graphics

Related to: GetSelectedInfo

Format: GetSelected(Object)

Parameters:

Object

Required. Any object expression for the window containing selected graphics.

Comments: This function may only appear in a script.

GetSelectedInfo

Description: Returns information about selected graphic item(s) in a window.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Graphics

Related to: CurrentWindow | GetSelected

Format: GetSelectedInfo(Object, Mode)

Parameters:

Object

Required. Any object expression for the window containing selected graphics.

Mode

Required. Any numeric expression that defines what information to return:

Mode	Information to return
0	Minimum left side user coordinate
1	Minimum bottom side user coordinate
2	Maximum right side user coordinate
3	Maximum top side user coordinate

Comments: If no graphic objects are selected, the return value will be meaningless.

Example:

```
sel = GetSelectedInfo(CurrentWindow(), 0);
```

This statement will set sel to the minimum left side (user) coordinate out of all selected graphic objects that are in the VTScada window that the mouse is over.

GetServer

(RPC Manager Library)

Description: Returns the name of the active server for a specified service.

Returns: Text

Usage: Steady State only.

Function Groups: Network

Related to: ConnectToMachine | DisconnectFromMachine | GetServersListed | GetStatus | IsClient | IsPotentialServer | IsPrimaryServer | Register (RPC Manager) | Send | SetRemoteValue | GetGUID

Format: \RPCManager\GetServer(ServiceName [, OptGUID])

Parameters:

ServiceName

Required. Any text expression giving the name of the service for which to get the active server's name.

OptGUID

An optional parameter that is any expression giving the 16-byte binary form of the globally unique identifier (GUID) for the application in which the service instance is located. The default is the application to which the caller belongs.

Comments: This module is a member of the RPC Manager's Library, and must therefore be prefaced by \RPCManager\, as shown in the "Format" section. If the application you are developing is a script application, the subroutine call must be prefaced by System\RPCManager\, and the System variable must be declared in AppRoot.src.
If the 16-byte binary format of the GUID is not known, the GetGUID function may be used to obtain it.

Example:

```
SName = \RPCManager\GetServer("ModemManager");
```

GetServerChanges

(RPC Manager Library)

Description: Launched by RPC Manager on a service server to obtain the service's synchronization data (i.e. called by RPC Manager during startup synchronization on a server to get the package of RPCs that create a synchronizable state on the client which is in step with the server).

Returns: See comments

Usage: Steady State only.

Function Groups: Network

Related to: See: "Adding Server-Only Synchronization" in the

Format: GetServerChanges(RevisionInfo, PackStreamRef, Client)

Parameters:

RevisionInfo

Required. The revision information from the GetClientRevision call made on the synchronizing client.

PackStreamRef

Required. A pointer to a variable to obtain changes.

ClientName

Required. The name of the client.

Comments: This subroutine is expected to be found in the object value of the caller of RPCManager\Register. It should not be called as "\RPCManager\GetServerChanges", but it may be appropriate to call it within the scope of the relevant service "<service scope>\GetServerChanges".
GetServerChanges() should not be written as a subroutine, as it will run on the RPCManager thread, thereby suspending external machine communication during synchronization.
If your GetServerChanges() request takes a long time, this will lead to a reduction in RPC throughput during RPC service synchronization, which may have undesirable effects. If GetServerChanges() is instead written as a launched module that slays itself when complete, the RPC thread will continue to service other RPC requests while you are building your synchronization data package.

Further, if GetServerChanges() is written as a subroutine, the RPC thread can be suspended while run-

ning your GetServerChanges() script, allowing your service thread to get a time slice and modify the service data that you are attempting to sample. This may cause two PCs running the same service to end up with mismatching data.

You must write GetServerChanges() as a launched module and call \RPCManager\SetDivert() as soon as you have finished sampling the synchronizable data for your service. If there is any possibility that another execution thread in your application can modify the data that GetServerChanges() is sampling during building of the synchronization package, you must protect the acquisition of the synchronization package with a CriticalSection().

GetServerMode

(RPC Manager Library)

Description: Returns the mode in which the current server for a specified service is running.

Returns: See comments

Usage: Steady State only.

Function Groups: Network

Related to:

Format: \RPCManager\GetServerMode(Service [, OptGUID]);

Parameters:

Service

Required. The name by which the service is known.

OptGUID

An optional parameter that is any expression giving

the 16-byte binary form of the globally unique identifier (GUID) for the application in which the service instance is located. The default is the application to which the caller belongs.

Comments: This subroutine is a member of the RPC Manager's Library, and must therefore be prefaced by \RPCManager\, as shown in the "Format" section. If the application you are developing is a script application, the subroutine call must be prefaced by System\RPCManager\, and the System variable must be declared in AppRoot.src.

The mode value represents the current synchronization state of the server for the specified service.

Presently, one of the following values may be returned

- \RPC_ACCEPT_ALL – the server is not performing synchronization with any client;
- \RPC_SYNC_MODE – the server is performing synchronization with a client; or
- \RPC_LINKCONTROL_ONLY – the server is starting synchronization with a client.

GetServerNumber

(RPC Manager Library)

Description: Returns the index down the prioritized server list of the current server for the specified service. Steady state or subroutine call.

Returns: Numeric

Usage: Steady State only.

Function Groups: Network

Related to:

Format: \RPCManager\GetServerNumber(Service [, OptGUID]);

Parameters:

Service

Required. The name by which the service is known.

OptGUID

An optional parameter that is any expression giving the 16-byte binary form of the globally unique identifier (GUID) for the application in which the service instance is located. The default is the application to which the caller belongs.

Comments: This subroutine is a member of the RPC Manager's Library, and must therefore be prefaced by \RPCManager\, as shown in the "Format" section. If the application you are developing is a script application, the subroutine call must be prefaced by System\RPCManager\, and the System variable must be declared in AppRoot.src.

The workstation at the top of the prioritized server list will return an index of zero.

GetServerSIDPtr

(RPC Manager Library)

Description: Returns a pointer to a variable that holds the session ID for the current server for the specified service.

Returns: Pointer

Usage: Steady State only.

Function Groups: Network

Related to:

Format: \RPCManager\GetServerSIDPtr(Service [, OptGUID]);

Parameters:

Service

Required. The name by which the service is known.

OptGUID

An optional parameter that is any expression giving the 16-byte binary form of the globally unique identifier (GUID) for the application in which the service instance is located. The default is the application to which the caller belongs.

Comments: This subroutine is a member of the RPC Manager's Library, and must therefore be prefaced by \RPCManager\, as shown in the "Format" section. If the application you are developing is a script application, the subroutine call must be prefaced by System\RPCManager\, and the System variable must be declared in AppRoot.src.

If the server changes, the value referenced by the returned pointer will change. This could be used by clients as a trigger to cause them to synchronize to the new server.

GetServersListed

(RPC Manager Library)

Description: This subroutine returns a one-dimensional array of the names or IPs of the servers that has been derived from the "-Servers" section of the service configuration file.

Returns: Array

Usage: Script Only.

Function Groups: Network

Related to: ConnectToMachine | DisconnectFromMachine | GetGUID | GetServer | GetStatus | IsClient | IsPotentialServer |

IsPrimaryServer | Register (RPC Manager) | Send | SetRemoteValue

Format: \RPCManager\GetServersListed(ServiceName [, OptGUID])

Parameters:

ServiceName

Required. Any text expression giving the name of the service for which to get the list of servers.

OptGUID

An optional parameter that is any expression giving the 16-byte binary form of the globally unique identifier (GUID) for the application in which the service instance is located. The default is the application to which the caller belongs.

Comments: This subroutine is a member of the RPC Manager's Library, and must therefore be prefaced by \RPCManager\, as shown in the "Format" section. If the application you are developing is a script application, the subroutine call must be prefaced by System\RPCManager\, and the System variable must be declared in AppRoot.src.
The return value from this subroutine is a list of the text names for all workstations listed as servers.
If the 16-byte binary format of the GUID is not known, the GetGUID function may be used to obtain it.

Example:

```
If 1 Main;
[
  sList = \RPCManager\GetServersListed("ModemManager");
]
```

GetServiceScope

(RPC Manager Library)

Description: Returns the service instance for a service.

Returns: See comments

Usage: Steady State only.

Function Groups: Compilation and On-Line Modifications, Network

Related to:

Format: \RPCManager\GetServiceScope(Service [, OptGUID]);

Parameters:

Service

Required. The name by which the service is known.

OptGUID

An optional parameter that is any expression giving the 16-byte binary form of the globally unique identifier (GUID) for the application in which the service instance is located. The default is the application to which the caller belongs.

Comments: This subroutine is a member of the RPC Manager's Library, and must therefore be prefaced by \RPCManager\, as shown in the "Format" section. If the application you are developing is a script application, the subroutine call must be prefaced by System\RPCManager\, and the System variable must be declared in AppRoot.src.

GetSessionContainers

Description: Returns an array of the names of tags that are "container" tags that exist at any level under the given context (parent) tag

Returns: Array

Usage: Script Only.

Function Groups: Variable

Related to: BuffOrder

Format: \ GetSessionContainers([Context, WithLocationOnly, ExcludeContextFromList]);

Parameters:

Context

Optional. The name of the tag under which to look for containers. Defaults to VTSDB.

WithLocationOnly

Boolean. When true, only tags that have valid Latitude and Longitude parameters will be returned

ExcludeContextFromList

Boolean. When true, the returned list will not include the tag specified in Context.

Comments: Returns Invalid if nothing is found.

GetSessionContainerTags

Description: Returns a dictionary of tag items below a given context (parent) tag.

Returns: Dictionary (see comments)

Usage: Script Only.

Function Groups: Variable

Related to: GetSessionContainers

Format: \ GetSessionContainerTags([Context, DoNotRecurseIntoContainer, NoContainersInTagsList, IncludeTagsInSubContainers, NavigationPath, PtrAutoDrilled]);

Parameters:

Context

Optional. The name of the tag under which to look for containers. Defaults to VTSDB.

DoNotRecurseIntoContainer

Optional Boolean. Set TRUE to not recurse into container with details page (default is false).

NoContainersInTagsList

Optional Boolean. Set TRUE to prevent containers from appearing in the Tags list. Defaults to FALSE.

IncludeTagsInSubContainers

Optional Boolean. Dpecifies whether or not to filter out descendents of containers. Normally defaults to false. Defaults to true if Context is an array or if Context tag has contributors

NavigationPath

Optional array. Unique ID values of the tags that make up the path taken to get to our Context tag.

PtrAutoDrilled

Optional pointer to an array of the Unique ID values of tags that were skipped (drilled through) to arrive at the Context tag. These will be skipped by the "Go Up" button in the site list user interface.

Comments: The returned dictionary will have the following structure:
["Title"] - The title to display for the returned data
["Tags"] - A list of tags that either exist under or contribute to the provided Context tag
["Containers"] - A list of Container tags that exist under the provided "Context" tag
Note that container tags use this function to obtain the list to be displayed in a Sites page. If the container tag also includes a module named CustomSiteListGetSubTags or CustomSiteMapGetSubTags, (which must be a subroutine and must return an array of tag names) then GetSessionContainerTags will automatically call that

module rather than GetTagList.
NavigationPath and PtrAutoDrilled are used only with Context tags that have an optional hook: CustomSiteListGetSubTags or CustomSiteMapGetSubTags.

GetSessionID

(RPC Manager Library)

Description: Returns the current session ID for a specified application on a workstation.

Returns: Numeric

Usage: Steady State only.

Function Groups: Network

Related to:

Format: \RPCManager\GetSessionID(NameOrIP [, OptGUID]);

Parameters:

NameOrIP

Required. Any of the names or IPs by which the workstation is known to the RPC Manager.

OptGUID

An optional parameter that is any expression giving the 16-byte binary form of the globally unique identifier (GUID) for the application in which the service instance is located. The default is the application to which the caller belongs.

Comments: This subroutine is a member of the RPC Manager's Library, and must therefore be prefaced by \RPCManager\, as shown in the "Format" section. If the application you are developing is a script application, the subroutine call must be prefaced by System\RPCManager\, and the System variable must be declared in AppRoot.src.

GetShapePath

Description: Returns the path value which defines the shape of a polygon.

Returns: Object

Usage: Script or steady state.

Function Groups: Graphics

Related to: GetModuleRefBox | GetPathBound

Format: GetShapePath(CodePointer)

Parameters:

CodePointer

Required. Any expression for the code pointer value that defines the graphic statement.

Example:

```
Start [
  If 1 Check;
  [
    drawWin = CurrentWindow() { Set drawing window };
    selObj = LastSelected(drawWin) { Get which graphic };
    graphicObj = GetShapePath(selObj) { Get shape of object };
  ]
]
Check [
  If Valid(graphicObj) EditGraphic;
  [
    UnselectGraphics(drawWin) { Use only this object };
    SelectPath(Self(), graphicObj) { Mark as selected };
  ]
]
```

The first state listed here retrieves the last selected graphic object and gets its shape in preparation for editing it. The next state then checks to make sure a valid graphic was chosen, then it releases all other chosen graphics in the window and selects the outline of the graphic as a precursor to editing it.

GetSocketStatus

(RPC Manager Library)

Description: Returns the connection status of either, 1) The machine node if the subnet is not valid, or 2) The socket that is on the specified subnet.

Warning: For use by advanced programmers only. Effective use of this function requires a thorough understanding of VTScada programming.

Returns: See comments

Usage: Steady State only.

Function Groups: Network

Related to: BinIP2Text | GetSocketStatus | TextIP2Bin

Format: \RPCManager\GetSocketStatus(MachineName [,Subnet])

Parameters:

MachineName

Required. The name of the machine for which you wish to get the Socket status.

Subnet

An optional parameter that specifies which subnet the socket you are interested in is on.

Comments: This subroutine is a member of the RPC Manager's Library, and must therefore be prefaced by \RPCManager\, as shown in the "Format" section. If the application you are developing is a script application, the subroutine call must be prefaced by System\RPCManager\, and the System variable must be declared in AppRoot.src.
This module returns the status of the connection between two PCs if no Subnet is specified. If the Subnet is specified, then the return value is that of the

Socket's status on the specified Subnet. If a Subnet is not specified and you are using a multi-homed PC, the return value will indicate how many sockets are connected to a particular PC.

Note: In a network using multi-homed PCs, it is possible for two PCs to be connected multiple times under a given Subnet. In such a case, GetSocketStatus will return the status of the first applicable socket found.

GetState

Description: Returns the code value for the specified state.

Warning: For use by advanced programmers only. Effective use of this function requires a thorough understanding of VTScada programming.

Returns: State code value

Usage: Script or steady state.

Function Groups: Compilation and On-Line Modifications, State

Format: GetState(Module, Name, Case)

Parameters:

Module

Required. Any expression for the code value that defines the module.

Name

Required. Any text expression which gives the text name of the state.

Case

Required. Any logical expression. If true, the name will be treated as case-sensitive. Otherwise the name will be treated as case-insensitive.

Comments: none

Related Functions:

GetInstance | GetStatement | GetStatementNum | GetStateText

GetStatement

Description: Returns the code value, statement offset or statement text size for the specified statement.

Returns: Varies. See Option parameter.

Usage: Script or steady state.

Function Groups: Compilation and On-Line Modifications, State

Related to: GetInstance | GetState | GetStatementNum | GetStateText

Format: GetStatement(Location, Index[, Option])

Parameters:

Location

Required. Any expression for the code value which defines the module and state.

Index

Required. Any numeric expression for the statement.

Option

Optional Boolean. If zero (the default) a suitable code value will be returned.
If set to 1, the statement offset in the source file will be returned.
If set to 2, the statement text size in the source file will be returned.

Comments: none

Example:

```
If 1 Check;
[
```

```
  dest = GetStatement(ActiveState(modPtr), 1);
]
```

This script gets the first statement in the active state of the module instance pointed to by modPtr.

GetStatementNum

Description: Returns the statement number for the specified statement.

Warning: For use by advanced programmers only. Effective use of this function requires a thorough understanding of VTScada programming.

Returns: Numeric

Usage: Script Only.

Function Groups: Compilation and On-Line Modifications, State

Related to: GetInstance | GetState | GetStatement | GetStateText

Format: GetStatementNum(Object, Statement)

Parameters:

Object

Required. Any expression for the module instance where the statement is located.

Statement

Required. Any expression for the code value or code pointer value which defines the statement. If this is an object value, the executing statement is used.

Comments: none

GetStateText

Description: Returns the text for the specified state.

Warning: For use by advanced programmers only. Effective use of this function requires a thorough understanding of VTScada programming.

Returns: Text

Usage: Script or steady state.

Function Groups: Compilation and On-Line Modifications, State

Related to: AdjustCode | GetModuleText | GetOneParmText | GetParmText | GetTransitText | GetVariableText | SetModuleText | SetOneParmText | SetParmText | SetStateText | SetTransitText | SetVariableText | TextOffset | TextSize

Format: GetStateText(State, Mode)

Parameters:

State

Required. Any expression for the code value which defines the state.

Mode

Required. Any numeric expression for the desired information:

Mode	Desired information
0	Character offset to beginning of state definition from start of module (includes state name and square brackets enclosing state code).
1	Size of state in characters, including state name and square brackets.
2	Character offset to beginning of first statement from start of state.
3	Size of state code in characters, excluding state name and square brackets.

Comments: none

Example:

```
If 1 Next;
[
  { The module and file for which to get state code }
  ModVal = FindVariable("ConfigFolder", Scope(\Code,
                        "AnalogInput"), 0, 0);
  FStream = FileStream("C:\VTScada\VTS\AnalogInput.SRC");
  { Get the states and create an array in which to store their code
}
  SList = StateList(ModVal, 1);
  Num = ArraySize(SList, 0);
  CodeList = New(Num);
  I = 0;
  WhileLoop(I < Num,
    StateOffset = GetStateText(SList[I], 0) +
  GetStateText(SList[I], 2).
    StateLength = GetStateText(SList[I], 3);
    { Read in the code contained in that state - don't include the
    square brackets }
    Seek(FStream, GetModuleText(ModVal, 1) + StateOffset, 0);
    StateCode[I] = BuffStream("");
    BlockWrite(StateCode[I], FStream, StateLength);
    Seek(StateCode[I], 0, 0);
    I++;
  );
  CloseStream(FStream);
]
```

GetStatus

(RPC Manager Library)

Description: Returns a variable that holds the current service instance status for the specified service.

Returns: Numeric

Usage: Steady State only. See: Rules for Usage.

Function Groups: Compilation and On-Line Modifications, Network

Related to: ConnectToMachine | DisconnectFromMachine | GetGUID | GetServer | GetServersListed | IsClient | IsPotentialServer | IsPrimaryServer | Register (RPC Manager) | Send | SetRemoteValue

Format: \RPCManager\GetStatus(ServiceName [, OptGUID])

Parameters:

ServiceName

Required. Any text expression giving the name of the service for which to get the connection status.

OptGUID

An optional parameter that is any expression giving the 16-byte binary form of the globally unique identifier (GUID) for the application in which the service instance is located. The default is the application to which the caller belongs.

Comments: This module is a member of the RPC Manager's Library, and must therefore be prefaced by \RPCManager\, as shown in the "Format" section. If the application you are developing is a script application, the module call must be prefaced by System\RPCManager\, and the System variable must be declared in AppRoot.src.
The return value from this subroutine is the current status of a service on the local workstation. It is similar to the return value for the Register module, except that in that case it is a pointer to the value that is returned rather than the value itself.
If the 16-byte binary format of the GUID is not known, the GetGUID module may be used to obtain it.

Example:

```
status = \RPCManager\GetStatus("ModemManager");
```

GetStreamLength

Description: Returns the present length of a stream in bytes.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Stream and Socket

Related to: BuffStream | ClientSocket | FileSize | FileStream |

PipeStream | Seek | ServerSocket | StreamEnd

Format: GetStreamLength(Stream)

Parameters:

Stream

Required. Any expression that returns a stream value.

Comments: This function is useful in determining the size of an existing stream. It is not necessary to do a Seek prior to executing the GetStreamLength.

Example:

```
sLength = GetStreamLength(BuffStream("abcde"));
```

This function will cause sLength to be set to 5.

GetStreamType

Description: Returns a type indication for a stream.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Stream and Socket

Related to: GetStreamLength

Format: GetStreamType(Stream)

Parameters:

Stream

Required. Any expression that returns a stream value.

Comments: This returns the type of an existing stream. Possible valid return values are as follows

Return Value	Meaning
0	file stream (temporary or persistent)
1	pipe stream
2	editor stream
3	buffer stream
4	TCP/IP socket stream
5	serial port stream
6	modem stream
7	TCP control stream

Invalid is returned for all non-stream values.

Example:

```
sType = GetStreamType(BuffStream("abcde"));
```

This function will cause sType to be set to 3.

GetSystemColor

Description: Returns the colors for the user-configured Windows™ colors.

Returns: Text (RGB color string)

Usage: Script or steady state.

Function Groups: Color, Graphics

Related to: PalStatus

Format: GetSystemColor(Option)

Parameters:

Option

Required. Any numeric expression that determines which color to return, as indicated in the following table.

Option	Color
0	Scroll bar
1	Desk top
2	Active window title bar
3	Inactive window title bar
4	Menu background
5	Window background
6	Window frame
7	Text in menus
8	Text in windows
9	Active window title text, size button, scroll bar arrow button
10	Active window border
11	Inactive window border
12	Background in MDI window
13	Background of selected control item
14	Text of selected item in control
15	Button face
16	Button shadow
17	Grayed text
18	Text on buttons
19	Inactive text title
20	Button highlight

Comments: This function is session aware. It will get VIC colors if called from within the VIC session.

Example:

```
ZText(25, 25, "Standard text", GetSystemColor(8), 0);
```

This puts text in the upper left corner of the screen in the standard Windows™ text color.

GetTagHistory

(Historian Manager Library)

Description: Launched module that retrieves historical data for a tag. Replaces GetLog.

Returns: Nothing

Usage: Script Only.

Function Groups: Log

Related to: GetLog

Format: \HistorianManager\GetTagHistory(PtrReturnCode, PtrResult, TagObj, FieldNames, StartTime, EndTime, TPP, NumEntries[, Modes, StaleTime, EnableDownTimeOverride, UseRecordOverrides])

Parameters:

PtrReturnCode

Required. A return code that may be numeric or may be a structure with an error code field. 0 indicates success. Non-zero indicates an error. If a structure is returned, it will be defined as:

```
STRUCT [
 ErrrorCode - An error code. See table in comments.
 ErrorText - Additional text to help diagnose the problem.
 ErrorTime - UTC timestamp of the error.
];
```

PtrReturnCode will be invalid until this launched function finishes.

PtrResult

A pointer to the historical data will be returned in this parameter.

TagObj

Object value of the tag for which the history is to be retrieved.

FieldNames

Either the name or, an array of names of the fields to retrieve.

StartTime

UTC time stamp indicating the beginning of the data range to retrieve.
A local time value should be converted to UTC as follows:
UTCStartTime = ConvertTimestamp(Start, LocalTimeZone, FALSE, Invalid);

EndTime

UTC time stamp indicating the end of the data range to retrieve. (Selection is inclusive of this time.)
Ignored if TPP is non-zero.

TPP

Required. Any numeric expression giving the time span in seconds for each array entry. Each array element will contain the data which correspond exactly to this time period which corresponds to 0 or more data points. If TPP is positive and FieldNames selects a text value, the first entry which falls in a time is read and Mode is ignored.
If TPP is equal to 0, the data is read and placed in the array on a one to one basis.
If TPP is less than 0, an error will be returned.

NumEntries

The number of log entries to be returned in the array. Use a negative value to retrieve values in reverse chronological order.

Modes

Optional numeric expression giving the method of handling the data. If TPP is greater than 0, the values that fall in each time span will be represented as follows:

Mode	Time Span Representation
0	Time weighted average
1	Minimum in range
2	Maximum in range
3	Change in value over the range
4	Value at start of range
5	Time of minimum in range (in seconds since Jan 1, 1970)
6	Time of maximum in range (in seconds since Jan 1, 1970)
7	Count the total number of zero to non-zero transitions within each TPP period.
8	Totals, for each TPP, the amount of time when the value is non-zero (Invalid is counted as zero).
9	Totals, for each TPP, the arithmetic sum of the recorded values.
10	Interpolates between values.
11	Difference between the start and end values of a range (see comments)

In the case of modes 5 and 6, FieldName should still be set to indicate the field on which the mode is to act; the return values will be times indicating the maximum or minimum in that field for each time span.

If TPP is less than or equal to 0, Mode is ignored. If the data is text, the first entry in a given time range is used for the array entry and Mode is ignored.

It is possible to retrieve more than one mode in a single GetTagHistory statement. To do this, pass an array of values in as the Mode parameter.

StaleTime

An optional parameter that sets a maximum validity duration for data elements that are being TPP processed. Normally, every data point is treated as remaining valid until the next data point. If a valid StaleTime parameter is given, then any data point will be treated as invalid StaleTime seconds after the recorded time. If TPP is less than or equal to 0, StaleTime is ignored. If StaleTime is not required but EnableDowntimeOverride is, then StaleTime should be given as an Invalid value.

It is possible to specify more than one stale time in a single GetTagHistory statement. To do this, pass an array of values in as the StaleTime parameter.

EnableDownTimeOverride

An optional Boolean. If valid, this will be used instead of the EnableDownTime property in the tag configuration.

UseRecordOverrides

An optional Boolean. Flag indicating if record overrides are in effect. Setting the UseRecordOverrides para-

meter to FALSE will cause all records in storage to be returned even if there are duplicate timestamps. Typically, a new record with a timestamp matching an existing record would be considered to override the existing record. The parameter defaults to TRUE, returning only the most recently added record for each timestamp.

Comments: This function is a replacement for GetLog. It has the following advantages over GetLog:

1) Works purely in UTC. Any conversion to local timestamps should be done by the caller only when absolutely necessary (only when they are about to be displayed or presented to an end-user).

2) Ambiguity removed from the StartTime, EndTime, TPP and NumEntries parameters: If TPP is non-zero, then EndTime is ignored and NumEntries explicitly specifies the number of periods to return.

3) Ambiguity removed from return values. There is an explicit return code set when the function is complete, rather than just returning a single-element array when an error occurs, which is ambiguous to whether or not it indicates an error or returned data. Moreover, the array returned is always a two-dimensional array.

Note: if multiple modes are being returned from the same field, the array size of fields needs to match that of mode.

When Mode is set to 11, GetTagHistory counts to a maximum value as set by StaleTime, and then rolls over to start a new interval. Whenever there is a decrease in value from one record to the next in the file, it is assumed that the maximum of the range has been reached and a rollover has occurred. Upon rollover, the value of GetTagHistory for the previous interval is calculated as (StaleTime + CurrentValue) – Minimum Value from previous interval.

Thus, for a StaleTime set to 100 and values read as follows: 10, 40, 90, 5.
The transition from 90 to 5 marks a rollover event. The value for the previous interval is (100 + 5) – 10 = 95.
This function forwards to GetLog if there is legacy history data involved in the query.
No downtime invalids are inserted when reading in the reverse order.
Defined error codes, which may be found in ptrReturnCode:

Constant	Value	Notes
HISTORIAN_SUCCESS	0	
HISTORIAN_ERROR_INVALID_ PARAMETER	1	
HISTORIAN_ERROR_UNKNOWN_ CONNECTION_TYPE	100001	
HISTORIAN_ERROR_ILLEGAL_ GETDATA_QUERY	100002	
HISTORIAN_ERROR_STORAGE_ LOCKED	100003	
HISTORIAN_ERROR_RECORDS_ DO_NOT_EXIST	100004	the records requested do not exist in storage according to the sequence counters
HISTORIAN_ERROR_ CONNECTION_FAILED	100005	
HISTORIAN_ERROR_RECORD_ READ_FAILURE	100006	the records requested should exist in storage, but there was an error retrieving them
HISTORIAN_ERROR_RECORD_ WRITE_FAILURE	100007	
HISTORIAN_ERROR_STORAGE_ INCONSISTENT_WITH_ SEQUENCE_COUNTER	100008	
HISTORIAN_ERROR_FIELD_	100009	

DOES_NOT_EXIST		
HISTORIAN_ERROR_RECORD_DELETION_FAILURE	100010	
HISTORIAN_ERROR_INITIALIZATION_FAILURE	100011	
HISTORIAN_ERROR_SCHEMA_PERSIST_FAILURE	100012	
HISTORIAN_ERROR_FAILED_TO_LAUNCH_THREAD	100100	

See Also:

WriteHistory, ConvertTimeStamp

Examples

There are two distinct ways to retrieve historical data: (1) Raw history and (2) Statistically processed data ("TPP").

A raw history retrieval retrieves the actual records that were logged by the tag

```
FieldNames = New(2);
FieldNames[0] = "Timestamp";
FieldNames[1] = "Value";
\HistorianManager\GetTagHistory(&RetCode, &Results, AnalogStatus1,
                        FieldNames, UTCStartTime, UTCEndTime,
                        0 {TPP=0 indicates raw records requested},
                         100 { Maximum number of records to return});
```

TPP-type query (average daily values for a 10-day period):

```
Modes = New(2);
Modes[0] = 4; { Timestamp at start of period }
Modes[1] = 0; { Time-weighted average of Value over the period }
\HistorianManager\GetTagHistory(&RetCode, &Results, AnalogStatus1,
                          FieldNames, UTCStartTime, 0 {EndTime
ignored},
                          86400 { 1-day time periods },
                          10    { 10 days requested  }, Modes);
```

Retrieve an average and a minimum:

```
Modes = New(2);
Modes[0] = 0 { Average };
Modes[1] = 1 { Minimum };
```

```
Fields = New(2);
Fields[0] = "Value";
Fields[1] = "Value";

\HistorianManager\GetTagHistory(&ReturnCode,&Result,TagObj,Fields,
                  StartTime,EndTime,EndTime - StartTime,1,Modes);
```

GetTagList

Description: Returns an array of tags, starting at a given point in the tag tree and including all child tags below that point, subject to the filtering parameters.

Returns: Array

Usage: Script Only.

Function Groups: Basic Module

Related to: PointList | GetTagTypes

Format: \GetTagList([RootLevel, DoRecursive, SearchType, SearchString, TagType, AreaSearch, EnableRealmAreaFiltering, IncludeGhosts, ExcludeLeaves])

Parameters:

RootLevel

Optional text. The name of the tag to begin with. May be a unique id or a tag name. Defaults to VTSDB if not specified.

DoRecursive

Optional. A Boolean value which, when set TRUE, results in sub-tags being returned as well as tags at the current level of the tree. Defaults to false if Invalid.

SearchType

Optional flag. Set 0 for a name search or 1 to search all parameters. Defaults to 0, name search.

SearchString

Optional text. A value to filter for, using pattern matching.

TagType

The type of tag to be filtered for.

AreaSearch

Optional text – the area name to filter for using pattern matching.

EnableRealmAreaFiltering

Optional Boolean. Set TRUE to enable area and realm filtering. Defaults to TRUE.

IncludeGhosts

Optional Boolean. Set TRUE to include disabled tags. Defaults to FALSE.

ExcludeLeaves

Optional Boolean. Set true to filter for only tags that have child tags. Defaults to FALSE.

Comments: This function provides the search and filtering features seen in the Tag Browser. You should select this function over the older PointList, since GetTagList adds several options to control what will be returned.

Examples:

Select all driver tags in application:

```
If 1;
[
   PList = \GetTagList(Invalid, TRUE, 0, "*", "Drivers", Invalid,
FALSE);
]
```

Select all the Analog Status tags, in all areas, within a context named MyStationName:

```
If 1;
[
   PList = \GetTagList("MyStationName", TRUE, 0, "*", "Ana-
logStatus");
]
```

GetTagTypes

Description: Returns an array of either the common names or the module names of all tag types. May optionally include the list of tag groups.

Returns: Array

Usage: Script Only.

Function Groups: Basic Module

Related to: PointList | GetTagList

Format: \GetTagTypes([GetGroups, GetModuleNames])

Parameters:

GetGroups

Optional Boolean. If set TRUE, the list of tag groups will be retrieved and returned at the end of the list of types. Defaults to TRUE.

GetModuleNames

Optional Boolean. If set TRUE, this function will return module names rather than the display name for each type.
If set, Groups will be invalid. Defaults to FALSE.

Comments: This function is used to populate a selection list of types.

Examples:

Select all driver tags in application:

```
If 1;
[
   PList = \GetTagList(Invalid, TRUE, 0, "*", "Drivers", Invalid,
FALSE);
]
```

Select all the Analog Status tags, in all areas, within a context named MyStationName:

```
If 1;
[
```

```
   PList = \GetTagList("MyStationName", TRUE, 0, "*", "Ana-
logStatus");
]
```

GetToken

Description: Reads the next token from a stream and returns the token type.

Warning: For use by advanced programmers only. Effective use of this function requires a thorough understanding of VTScada programming.

Returns: Numeric (see table in comments)

Usage: Script Only.

Function Groups: Compilation and On-Line Modifications, Stream and Socket

Related to: Compile

Format: GetToken(Stream, Token, ClassBuffer, NumClasses, StateBuffer, ActionBuffer, SkipWhite, LineCount, Column, CharCount)

Parameters:

Stream

Required. Any stream expression for the input stream.

Token

Required. A variable into which the next token read will be stored.

ClassBuffer

Required. Any text expression for the character classifier table.

NumClasses

Required. Any numeric expression for the number of character classes.

StateBuffer

Required. Any text expression for the tokenizer state

table.

ActionBuffer

Required. Any text expression for the tokenizer action table.

SkipWhite

Required. Any logical expression. If true, all white spaced is skipped. Otherwise all continuous white space is treated as a token.

LineCount

Required. Must be a variable. The number of lines read is stored here.

Column

Required. Must be a variable. The current column is stored here.

CharCount

Required. Must be a variable. The number of characters read is stored here.

Comments: The return value for this function is the token type. The tokens recognized by the Compile function are as follows (assuming the tables used by the compiler are those handed in to this function)

Return Value	Token Type
0	End-of-file
1	White space
2	Identifier - variable

GetTrajectoryPath

Description: Returns the Path value which defines the trajectory of a graphic object.

Returns: Path

Usage: Script or steady state.

Function Groups: Graphics

Related to: GetPathBound | Trajectory

Format: GetTrajectoryPath(CodePointer)

Parameters:

CodePointer

Required. Any expression for the code pointer value that defines the graphic statement.

GetTransform

Description: Returns the transform value applied to a graphic statement.

Returns: Transform

Usage: Script or steady state.

Function Groups: Graphics

Related to: GetXformRefBox | GUITransform | UnTransform

Format: GetTransform(CodePointer)

Parameters:

CodePointer

Required. Any expression for the code pointer value which defines the graphic statement.

GetTransitText

Description: Get Transition Document Text. This function returns information about the documentation of an action.

Warning: For use by advanced programmers only. Effective use of this function requires a thorough understanding of VTScada programming.

Returns: Text

Usage: Script Only.

Function Groups: Compilation and On-Line Modifications, Advanced Module

Related to: AdjustCode | GetModuleText | GetOneParmText | GetParmText | GetStateText | GetVariableText | SetModuleText | SetOneParmText | SetParmText | SetStateText | SetTransitText | SetVariableText | TextOffset | TextSize

Format: GetTransitText(Action, Mode)

Parameters:

Action

Required. Any expression for the code value that defines the action.

Mode

Required. Any numeric expression which defines the information desired

Mode	Information desired
0	Script size in characters
1	Character offset to script
2	Trigger size in characters
3	Character offset to trigger
4	Destination size in characters
5	Character offset to destination
6	Size of script excluding [ ] , in characters
7	Character offset to first script statement
8	Total size of action, in characters

GetUserID

Description: Returns the name of the user for the current session.

Returns: Text

Usage: Script only.
May be used in optimized Tag Parameter Expressions.

Function Groups: Variable

Related to: GetUserSession

Format: Layer\GetUserID()

Parameters: none

GetUserName

Security Manager Module

Description: Returns the user name of the caller's account.

Returns: String

Usage: Script or steady state.

Related to: GetAccountID | GetAccountInfo | GetFullName | GetGroupName | IsLoggedOn | IsSecured | IsSuspended | SecurityCheck | UIErrorToText

Format: \SecurityManager\GetUserName()

Parameters: None

Comments: This is not namespace-qualified.

GetUserNameOfRecord

(Alarm Manager module)

Description: Given an alarm record, returns the user name associated with the transaction.

Returns: Text

Usage: Script Only.

Function Groups: Alarm

Related to: GetAlarmObject

Format: \AlarmManager\GetUserNameOfRecord(AlarmRecord[, TrimRealm])

Parameters:

AlarmName

Required record. A reference to the alarm, as returned by the function GetAlarmObject.

TrimRealm

Optional Boolean. If TRUE, and if the user is a member of a security group, the realm (group) will be trimmed from the result.

Comments: This function is intended primarily for presentation purposes.

Example:

Related Information:

GetUserSession

Description: The module this function returns is useful for accessing session-specific variables. The function will traverse the call tree to find the session that called it.

Returns: A session module instance.

Usage: Script or steady state.

Function Groups: Basic Module, Variable

Related to: ParentWindow

Format: \GetUserSession([DefaultToRootSession])[\QueryVariable]

Parameters:

DefaultToRootSession

Optional Boolean. If TRUE (the default value) then if a valid session is not found, this function will return the Display Manager's root session. (That is, the one pertaining to the actual running application as opposed to any remotely established session.)
If set FALSE, then Invalid will be returned if a valid session is not found.

QueryVariable

Optional. The name of a variable to be queried.
Examples include, but are not limited to:
\IsRootSession
\AppTitle
\CurrentWinInst\PageInstance

Comments:

This routine walks up the caller's call tree, looking for a variable named "_UserSession_", then returning the context containing that variable. In none is found, then the Display Manager's root session will be return unless the optional parameter is set to FALSE.
GetUserSession is also useful for launching dialogs in the scope of a particular user session. This means that a tag or service which is not associated with any particular user can cause a dialog to open on a particular VIC session.
Note that the public variable, IsEditing, is obsolete. In place of GetUserSession\IsEditing, new code should use ParentWindow()\Editing.

Examples:

Launch a dialog in the scope of a session:

```
If OpenDialog Idle;
[
  Session = \GetUserSession();
```

```
  Launch("Dialog", Self() { Parent }, Session { Caller });
]
```

GetValue

(Hierarchical Accumulator module)

Description: Returns the current count of the values within an accumulator dictionary.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Variable

Related to: Accumulate

Format: HierarchicalAccumulator\GetValue(TagObj, AccumulatorName);

Parameters:

TagObj

Required. The tag object at the point in the hierarchy where you want to collect the accumulated values.

AccumulatorName

Required. The name of the accumulator, from which to retrieve the current count.

Comments: This function is part of the HierarchicalAccumulator module, so must always be called as shown in the format. You will need this function if you have created your own accumulator and wish to retrieve the value.

The accumulator allows a fresh count to be generated at different levels in a tag tree, and as tags are moved or disabled.

Examples:

```
Main [
   Return(\HierarchicalAccumulator\GetValue(ContainerObj, "AlarmUn-
acked"));
]
```

GetVariableText

Description: Returns information about a variable.

Warning: For use by advanced programmers only. Effective use of this function requires a thorough understanding of VTScada programming.

Returns: Numeric

Usage: Script Only.

Function Groups: Compilation and On-Line Modifications, Variable

Related to: AdjustCode | GetModuleText | GetOneParmText | GetParmText | GetStateText | GetTransitText | SetModuleText | SetOneParmText | SetParmText | SetStateText | SetTransitText | SetVariableText | TextOffset | TextSize

Format: GetVariableText(Variable, Mode)

Parameters:

Variable

Required. Any expression for a variable.

Mode

Required. Any numeric expression for the desired information. The character offset is to the right after the declared variable (i.e. the carriage return is not included, nor is the line feed that ends the line), and length includes the CR, LF(2).

Mode	Desired information
0	Character offset to variable declaration
1	Size of variable declaration in characters

In the case of the offset to the variable declar-

ation, this is located immediately following the variable preceding this variable and will in fact include the carriage return and line feed from the previous variable. Likewise, the size of the variable declaration includes the leading carriage return and line feed, any spaces preceding the actual variable name and does not include the carriage return and line feed following the variable.

Example:
Assume that the following text is the entire document file for a module:

```
[
  x;
  y;
  [ (99)
    P1 Module "P1.SRC";
  ]
]
Main [
]
```

The offsets for the variables would be as follows:

Variable	Offset (Mode = 0)	Size (Mode = 1)
X	1	6
Y	7	6
P1	23	25

GetVariableType

Description: Returns the type, BASEVALUE, stored within a variable.

Warning: For use by advanced programmers only. Effective use of this function requires a thorough understanding of VTScada programming.

Returns: Integer or Structure.

Usage: Script Only.

Function Groups: Variable

Related to: SetVariableType

Format: GetVariableType(Variable)

Parameters:

Variable

Required. Any expression for a variable.

Comments: If a structure is returned, the first element will be an integer giving the data type. In the case where the variable is a module, the remaining elements will be text strings giving the names of modules within the scope.

Example:

```
Var = FindVariable(ParmList[I], Mod, 0, 0);
{ Retrieve any typing information }
ParmType = GetVariableType(Var);
```

GetVarMetadata

Description: Every variable object contains an embedded value. This function is used to retrieve those values.

Warning: For use by advanced programmers only. Effective use of this function requires a thorough understanding of VTScada programming.

Returns: Varies

Usage: Script Only.

Function Groups: Dictionary, Variable

Related to: SetVarMetadata | FindVariable | AddVariable

Format: GetVarMetadata(Variable)

Parameters:

Variable

Required. A variable handle, such as would be

returned from the FindVariable or AddVariable functions.

Comments: Commonly used in conjunction with SetVarMetadata, FindVariable or AddVariable. Note that type data for each variable is stored within the variable using metadata.

Example:

```
<
TestMod
[
  X;
  Y;
  Var;
]
Main [
  If ! Valid(X);
  [
    X = "This is the value of X"
    Var = FindVariable("X", Self(), 0, 0);
    SetVarMetadata(Var, "This is the metadata in variable X");
    Y = GetVarMetadata(Var);
  ]
  ZText(50, 100, Concat("X: ", X), 14, 0);
  ZText(50, 120, Concat("Y: ", Y), 14, 0);
]
>
```

GetVoices

(VoiceTalk Module)

Description: Runs in the VoiceTalk thread and returns a list of voices available on a SAPI text-to-speech stream.

Returns: Array

Usage: Script Only.

Function Groups: Speech and Sound

Related to: Configure | GetDevices | Reset | ShowLexicon | Speak | VoiceTalk

Format: VoiceTalkStream\GetVoices([Detailed])

Parameters:

VoiceTalkStream

Required. A speech stream returned from VoiceTalk.

Detailed

Required. A flag which, if set to a non-zero value, results in a detailed 2-dimensional array of information about the voices being returned. If this parameter is "0" or omitted, the return value will be a 1-dimensional list of voice names suitable for use in the VoiceTalk\Configure function.
If Detailed is specified, a 2-dimensional array containing the available speech voices will be returned. Each row in the array represents a speech module as follows:

[N][0] Text value that indicates the name of this voice This is a human readable string (such as "Microsoft Mary") that may be passed to the VoiceTalk\Configure function.

[N][1] Text value giving the language of the voice. For example, American English will be returned as "409,9", where 409 is the hexadecimal representation of standard English (decimal 1033), and 9 indicates American version of English.

[N][2] Text value giving the gender of the voice. Either "Male" or "Female"

[N][3] Text value giving the age of the voice (e.g., "Adult")

[N][4] Text value vendor providing the voice (e.g. "Microsoft")

Comments This function will return immediately. Using the array

information returned from this call, it is possible to determine whether a particular SAPI text-to-speech mode exists on the system. If it does not, the array information can be used to select the voice that most closely matches the desired characteristics, or build and display a dialog to allow the user to choose.

Example:

```
sHandle = \VoiceTalk();
If Valid(sHandle) && ! getVoices;
[
  getVoices = 1;
  sVoices = sHandle\GetVoices(1);
]
```

This will return a detailed 2-dimensional array of all available text-to-speech engine voices in the array, sVoices.

GetWCPath

Description:	Returns the file system path to the application's working copy folder.
Returns:	Text
Usage:	Steady State only. See: Rules for Usage.
Function Groups:	Configuration Management
Related to:	GetOEMLayer |
Format:	Layer\GetWCPath
Parameters:	none
Comments:	The retrieved path includes the trailing backslash.

Examples:

This snippet from a script application will display the working copy path for the completed tutorial. (The GUID of the completed tutorial may vary.)

```
[
  WaitObj;
  CompLayer;
  TutGUID = "db53f244-90ef-4628-bdf6-2d53794a2079";
]
Main [
  If !Valid(WaitObj) && !Valid(TestLayer) WaitTestLayerLoad;
  [
    { GetAppInstance doesn't return the Layer until it has loaded. }
    WaitObj = Layer\GetAppInstance(TestLayerGUID, &TestLayer);
  ]
]
WaitTestLayerLoad [
  WC = TestLayer\GetWCPath();
  ZText(100, 100 { Lower left corner of text },
    WC { Text to display },
    0 { Text is black },
    0 { Use default font });
]
```

GetWCRevision

Description: Returns the revision structure for the repository revision in use by the working copy.

Returns: Revision Node.

Usage: Script Only.

Function Groups: Configuration Management

Related to: GetWCPath |

Format: LayerRoot\GetWCRevision()

Parameters: None

Comments: None

Examples:

GetXformRefBox

Description: Get Transform Reference Box. This function returns the reference box for any transform of a module.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Compilation and On-Line Modifications, Graphics, Advanced Module, Window

Related to: GetModuleRefBox | GUITransform | UnTransform

Format: GetXformRefBox(Object, Option)

Parameters:

Object

Required. Any expression which gives the object value for the transformed module instance.

Option

Required. Any expression that defines the requested return value as indicated by the following:

Option	Return Value
0	Left side
1	Bottom side
2	Right side
3	Top side

Comments: This function will only return the reference box of a transform immediately acting upon the module indicated by Object, it will not search for a transform that may be acting upon the module's parent or ancestors.
If the module indicated by Object is not under the influence of a transform, the return value will be Invalid.
This function is unaffected by the Untransform statement. Even if the module has been untransformed, it will still be able to retrieve the reference box of a transform that would be otherwise acting upon it.

Example:

Suppose that the System module calls module Motor inside of a transform as follows:

```
GUITransform(0, 100, 100, 0 { Bounding box for transform },
             1, 1, 1, 1, 1 { No scaling },
             0, 0 { No trajectory or rotation },
             1 { Module graphics are visible },
             0 { Reserved },
             0, 0, 0 { Cannot be focused/selected },
             Motor() { Module to transform });
```

Further suppose that Motor needs to know its exact location in the window - it is inside of the transform and is therefore unaware of its (the transform's) parameters. This can be done by the following:

```
right = GetXFormRefBox(Self(), 2);
```

The variable right will contain the X-coordinate for the right of the transform's reference box, which in this case will be 100. If the transform's position in the window were to change, the value of this variable would similarly change.

GetXMLNodeArray

(System Library)

Description: Searches the result returned from XMLParse and returns an array of XMLNode values of a given type. Returns Invalid if no matches are found.

Returns: Array of XMLNodes.

Usage: Script Only.

Function Groups: XML

Related to: XMLProcessor | XMLAddSchema | XMLWrite | XMLCloneNode | XMLCreateNode | XMLDeleteMember | XMLGetNode | XMLParse

Format: \System\GetXMLNodeArray(XMLContainer, ElementType)

Parameters:

XMLContainer

The XMLNode within which to search for element type name. Only the direct child nodes of this node are searched.

This is the equivalent of the first parameter of the Scope operator.

ElementType

The element type name to search for.
This is equivalent to the second parameter of the Scope operator.

Comments: This helper function was created to work with the results of XMLParse.
GetXMLNodeArray searches the immediate children of a given XMLNode value (first parameter) for elements with the type name provided by the second parameter and returns those elements in an array of XMLNodes. This serves to simplify parsing of XMLNode trees, which may contain lists of repeating elements.
Elements in the returned array will be in the same order as in they were in the original XML document.
If no elements with the given name are found, then this function will return INVALID.

Examples:

```
Source = "<X><Y>"Hello"</Y><Y>" "</Y><Y>"world"</Y><Y>"!"</Y></X>";
XMLProc = XMLProcessor(INVALID);
XMLParse(XMLProc, Source, Errors, Document);
RootNode = XMLGetNode(Scope(Document, "X"));
TextNodes = \System\GetXMLNodeArray(RootNode, "Y");
At this point TextNodes will contain an array of four XMLNodes, con-
taining "Hello", " ", "world", and "!" respectively.
```

GoToOffset

Note: Deprecated. Do not use in new code.

Description: Forces an editor to move to a location in its text.

Returns: None

Usage: Script or steady state.

Function Groups: Editor

Related to: AddEditorText | CurrentLine | Editor | ForceEvent | MakeEditor | SetEditMode

Format: GoToOffset(Editor, Offset, Highlight, NumHighlight)

Parameters:

Editor

Required. Any expression for the editor value.

Offset

Required. Any numeric expression for the character offset to display.

Highlight

Required. Any logical expression. If true, characters will automatically be highlighted, beginning at Offset.

NumHighlight

Required. Any numeric expression for the number of characters to highlight if Highlight is true.

Example:

```
myEditor = MakeEditor() { Create the editor };
...
GoToOffset(myEditor { Which editor to use },
           100 { Move to an offset of 100 bytes },
           1 { Highlight characters at new offset },
           10 { Highlight 10 characters });
```

Grid

Description: Places a (lined) grid pattern on the screen.

Returns: Nothing

Usage: Steady State only.

Function Groups: Graphics

Related to: Box | GUIRectangle| Line | ZBox | ZGrid

Format: Grid(X1, Y1, X2, Y2, Style, Color, Horizontal, Vertical)

Parameters:

X1

Required. Any numeric expression giving the X coordinate on the screen of one side of the grid area.

Y1

Required. Any numeric expression giving the Y coordinate on the screen of either the top or bottom of the grid area.

X2

Required. Any numeric expression giving the X coordinate on the screen of the side of the grid area opposite to X1.

Y2

Required. Any numeric expression giving the Y coordinate on the screen of either the top or bottom of the grid area, whichever is the opposite to Y1.

Style

Required. Any numeric expression giving the line style for the grid lines. Valid line styles are from 1 to 10, where 1 is a solid line.

Color

Required. Any numeric expression giving the color of the grid lines. If the number is less than 10000, the grid lines are non-destructive. If greater than or equal to 10000, the grid lines are destructive and the actual color used is Color - 10000. RGB values and system color constants are not supported.

Horizontal

Required. Any numeric expression giving the number of horizontal lines in the grid. This value may be zero.

Vertical

Required. Any numeric expression giving the number

of vertical lines in the grid. This value may be zero.

Comments This statement is non-destructive unless Color is explicitly set destructive. The outside perimeter of the grid area is not drawn.

Example:

```
Grid(100, 100, 700, 500 { Bounding box for the grid },
     3 { Dotted line style },
     9 { Light blue color },
     3 { 3 horizontal lines divide 4 areas },
     5 { 5 vertical lines divide 6 areas });
```

This example divides an area into 6 blocks wide by 4 blocks high using 3 horizontal lines and 5 vertical lines.

GridList

(System Library)

Description: Draws a list in the style of a spreadsheet.

Returns: Nothing

Usage: Steady State only.

Function Groups: Graphics

Related to: CheckBox | Droplist | GUITransform | HScrollbar | Listbox | RadioButtons | Spinbox | SplitList | ToolBar | VScrollbar |

Format: \System\GridList(Titles, Data, DataFormat, ColWidthsParm, NumDataRowsParm, NumDataColsParm, GridListBGndParm, GridColorParm, GridLineWidthParm, GridStyleParm, RowHeightParm, TitleHeightParm, HCellPaddingParm, VCellPaddingParm, HScrollPosParm, VScrollPosParm, DisableVScroll, DisableHScroll, DisableColumnSizing, DisableSorting, DisableSelectedCell, DisableVGridLines, DisableHGridLines [, LockFirstColumn , Sort, SelectedRow, SelectedColumn, GridFontParm, GetSortKeyScope, EnableBorderParm])

Parameters:

Titles

Required. The array of the titles you wish to use for the grid column headings. The title bar for a grid can be disabled by setting the Titles array to Invalid, or by setting TitleHeightParm to 0. This (and all other) array parameters must use dynamic arrays.

Alternatively, you may provide the name of a call-back module, which will provide the titles. The module must allow for sorting by title.

Data

Required. The array of data with which to populate the grid.

DataFormat

Required. The array of the data formats corresponding to the data specified in the Data array. There must be one entry per column of data.

ColWidthsParm

Required. The array of the widths of the columns in pixels.

NumDataRowsParm

Required. Specifies the number of data rows to display.

NumDataColsParm

Required. Specifies the number of data columns to display.

GridListBGndParm

Required. Indicates the background color for the cells in the grid.

GridColorParm

Required. Indicates the color of the grid lines.

GridLineWidthParm

Required. Indicates the width of the grid lines.

GridStyleParm

Required. Indicates the style of the grid lines.

RowHeightParm

Required. Indicates the height of the rows in the grid.

TitleHeightParm

Required. Indicates the height of the column headings row above the grid. The title bar for a grid can be disabled by setting TitleHeightParm to 0, or by setting the Title array to Invalid.

HCellPaddingParm

Required. Indicates the horizontal cell padding for the grid.

VCellPaddingParm

Required. Indicates the vertical cell padding for the grid.

HScrollPosParm

Required. Indicates the horizontal scroll bar position.

VScrollPosParm

Required. Indicates the vertical scroll bar position.

DisableVScroll

Required. A flag that may be set to TRUE (non-zero) to disable vertical scrolling, or FALSE (0) to enable vertical scrolling.

DisableHScroll

Required. A flag that may be set to TRUE (non-zero) to disable horizontal scrolling, or FALSE (0) to enable horizontal scrolling.

DisableColumnSizing

Required. A flag that may be set to TRUE (non-zero) to

disable column resizing, or FALSE (0) to enable column resizing.

DisableSorting

Required. A flag that may be set to TRUE (non-zero) to disable sorting by clicking the column headings, or FALSE (0) to enable sorting by clicking the column headings.

DisableSelectedCell

Required. A flag that may be set to TRUE (non-zero) to disable selected cell highlighting, or FALSE (0) to enable selected cell highlighting.

DisableVGridLines

Required. A flag that may be set to TRUE (non-zero) to disable vertical grid lines, or FALSE (0) to enable vertical grid lines.

DisableHGridLines

Required. A flag that may be set to TRUE (non-zero) to disable horizontal grid lines, or FALSE (0) to enable horizontal grid lines.

LockFirstColumn

An optional flag that may be set TRUE (non-zero) to lock the first column from horizontal scrolling (e.g. the first column contains row titles). The default value if Invalid or not supplied is FALSE.

Sort

An optional parameter that specifies the column on which the GridList is to be sorted. The parameter is 1-based (i.e. a value of 1 refers to the first column).
If negative, the sort order is descending If the user clicks a column title to sort the GridList, then the parameter is set to the appropriate value. .
If Invalid or not supplied, no column sorting will be applied.

SelectedRow

An optional parameter that specifies the row of the selected cell. The parameter is 0-based (i.e. a value of 0 refers to the first row). If the user clicks in a cell, then the parameter is set to the index of the selected row.
If Invalid or not supplied, no cell is selected.

SelectedColumn

An optional parameter that specifies the column of the selected cell. The parameter is 0-based (i.e. a value of 0 refers to the first column). If the user clicks in a cell, then the parameter is set to the index of the selected column.
If Invalid or not supplied, no cell is selected.

GridFontParm

An optional parameter that specifies the font to be used for Titles and Grid elements. The selected font will affect all items in the GridList.
If using callbacks, it will allow the user to use the same font in their callbacks or have a separate title font and use their own font in their callbacks.
Note: Nothing in a GridList is sized based on font size. The caller must ensure that the row and title heights are large enough to accommodate the font.
No default value.

GetSortKeyScope

An optional module value of GetSortKey call-back (example in comments section). Required only when using call-backs for the cells of the gridlist.

EnableBorderParm

Optional Boolean. Set true to show a border around the grid. Defaults to FALSE.

Comments: GridList is a member of the VTS System Library, and must therefore be prefaced by "\System\", as shown

in the "Format" section above. If you are developing a script application, use "System\..." rather than "\System\..." in the function call.
Where parameters use arrays, they must be dynamic arrays.

Using GridList, you may enable such functionality as column resizing, sorting when clicking on column headings, selection of cells, using the keyboard to move around the grid, and adding scroll bars.

GridList also provides the ability to define a call-back function to draw in a cell

```
Data[I]     = Self()      { Module in which the
CallBack exists };
DataForms[I] = "DrawCell"  { The CallBack module
used to display a cell
in this column };
```

An example of a GetSortKey call-back module:

```
GetSortKey
(
  Row                 { Row index to get the
sort key for                 };
  Column              { Column index to get the
sort key for              };
  Inverted            { Bool: TRUE if the order
is being reversed         };
)
Main [
  If 1;
  [
     Return(ToUpper(Data[Row][Column]);
  ]
]
```

Note: GridList has been configured to clip text on the right and display a tooltip if there is not enough room to show the entire text string in a cell.

Example:

(See also, an example in GUITransform)

```
   {initialize the data array}
   Titles[0] = "Name";
   Titles[1] = "Area";
   Titles[2] = "Description";
   Titles[3] = "I/O Device";
   Titles[4] = "Address";
   Data[0][0] = "Tag1";
   Data[1][0] = "Tag2";
   Data[2][0] = "Tag3";
   Data[0][1] = "Area1";
   Data[1][1] = "Area1";
   Data[2][2] = "I'm a tag";
   Data[1][3] = "PollDrvr1";
   Data[0][4] = "40001";
   DataForm  = "%s";

GUITransform(50, 500, 350, 50  { Reference rectangle         },
               1                { Scale Left                 },
               1                { Scale Bottom               },
               1                { Scale Right                },
               1                 { Scale Top                  },
               1                 { No overall scaling         },
               0, 0, 1, 0        { No movement; visible; res  },
               0, 0, 0           { Not selectable             },
               \System\GridList(Titles  { Titles array        },
                                Data   { Data array          },
                                DataForm { Cell format array  },
                                60    { Column widths array  },
                                3      { # Data rows          },
                                5      { # Data cols          },
                                15, 8  { Grid BGnd, Grid color },
                                1, 1 { Grid line width, line style},
                                30, 30 { Row/Title height  },
                               0, 0 { Horiz/Vert cell padding },
                                0, 0 { Horiz/Vert Scroll position },
                                1, 1 { Disable V/H scroll bars },
                                1    { Disable column sizing   },
                                1    { Disable Sorting         },
                                1    { Disable selected cell   },
                                0, 0 { Disable V/H grid lines  },
                                1    { LockFirstColumn         },
                                Invalid  { Sort                },
                                0     { SelectedRow         },
                                0      { SelectedColumn      },
                                \_DialogFont { GridFontParm    }));
```

This code example will create a Grid List that looks like the following:

Name	Area	Description	I/O Device	Address
Tag1	Area1			40001
Tag2	Area1		PollDrvr1	
Tag3		I'm a tag		

GUIArc

Description: Draws an arc in a window. Can return a Boolean when selected by a mouse button or when the <ENTER> key is pressed after the graphic acquires focus.

Returns: Numeric

Usage: Steady State only.

Function Groups: Graphics

Format: GUIArc(LeftReference, BottomReference, RightReference, TopReference, ScaleLeft, ScaleBottom, ScaleRight, ScaleTop, ScaleWhole, Trajectory, Rotation, Opacity, Reserved, Button, FocusID, FocusTrigger, Pen, Vertex)

Parameters:

LeftReference

A constant number that gives the left side reference coordinate. It must be a constant. A variable or expression is not valid here.

BottomReference

A constant number that gives the bottom side reference coordinate. It must be a constant. A variable or expression is not valid here. The top and bottom references are measured down from the top of the screen. The top and bottom references are measured down from the top of the screen.

RightReference

A constant number that gives the right side reference coordinate. It must be a constant. A variable or expression is not valid here.

TopReference

A constant number that gives the top side reference coordinate. It must be a constant. A variable or expression is not valid here.

ScaleLeft

Required. Either a numeric expression, or any expression that returns a Normalize value. This parameter scales this side from its reference position with respect to the opposite side. If it is a numeric expression, a value of 1 will place the side at its reference position. A value of 0 will place it at the opposite side reference position. Similarly, a Normalize value will scale the side between the high and low limits. If the value is at the high level, the side will be at its reference position. If the value is at the low level, the side will be at the opposite side reference position.

ScaleBottom

Required. Either a numeric expression, or any expression that returns a Normalize value. This parameter scales this side from its reference position with respect to the opposite side. If it is a numeric expression, a value of 1 will place the side at its reference position. A value of 0 will place it at the opposite side reference position. Similarly, a Normalize value will scale the side between the high and low limits. If the value is at the high level, the side will be at its reference position. If the value is at the low level, the side will be at the opposite side reference position.

ScaleRight

Required. Either a numeric expression, or any expression that returns a Normalize value. This parameter scales this side from its reference position with respect to the opposite side. If it is a numeric expression, a value of 1 will place the side at its reference position. A value of 0 will place it at the opposite side reference position. Similarly, a Normalize value will scale the side between the high and low limits. If the value is at the high level, the side will be at its reference position. If the value is at the low level, the side will be at the opposite side reference position.

ScaleTop

Required. Either a numeric expression, or any expression that returns a Normalize value. This parameter scales this side from its reference position with respect to the opposite side. If it is a numeric expression, a value of 1 will place the side at its reference position. A value of 0 will place it at the opposite side reference position. Similarly, a Normalize value will scale the side between the high and low limits. If the value is at the high level, the side will be at its reference position.

If the value is at the low level, the side will be at the opposite side reference position.

ScaleWhole

Required. Either a numeric expression, or any expression that returns a Normalize value. This parameter scales the horizontal and vertical dimensions by the specified factor before the left, bottom, right and top coordinates are scaled.

Trajectory

Required. Either a Trajectory function, a variable containing a Trajectory value, or a numeric expression. If this is a Trajectory value or function, the appropriate translation is applied to the image after the rotation is applied. If it is a valid numeric expression, the image isn't translated, but is displayed. Any other value is Invalid.

Rotation

Required. Either a Rotate function, a variable containing a Rotate value, or a numeric expression. If this is a Rotate value or function, the appropriate rotation is applied to the image before the trajectory is applied. If it is a valid numeric expression, the image is rotated clockwise the number of degrees specified. Any other value is Invalid.

Opacity

Required. Any Numeric expression, setting the opacity of the object. A value of one results in a solid, zero is invisible and values between zero and one are used as an alpha setting for opacity.

Reserved

Reserved for future use, set to 0.

Button

Required. Any numeric expression giving the button combination that activates this graphic.

Button	Locator Buttons
0	No button combination will activate this graphic
1	Right button
2	Middle button
3	Right and middle buttons
4	Left button
5	Left and right buttons
6	Left and middle buttons
7	All three buttons

If the above values are multiplied by 8, the meaning for multiple buttons pressed becomes "OR" rather than "AND." For example, to accept any button on a 2 or 3 button mouse, use 56 (i.e. 8 * 7). To accept the left mouse button regardless of whether or not the right button is pressed, use 32 (i.e. 8 * 4).

If a 64 is added to this parameter, the function will become true when the mouse buttons are released rather than when they are pressed.

FocusID

Required. Any numeric expression giving the focus number of this graphic. If FocusID is zero, this graphic cannot receive the input focus. This parameter's value may be used in a NextFocusID statement to force this graphic to get the focus.

FocusTrigger

Required. Any Boolean expression. If FocusTrigger

changes from a valid false to a valid true, this graphic will become acquire focus.

Pen

Required. Any expression that returns a Pen value to describe the color, width and style. You may also use any of the following to draw a single-pixel, solid arc:

- a palette index VTScada Color Palette
- a system color Constants for System Colors
- an RGB string with optional Alpha value in the format, "<AARRGGBB>", or "<RRGGBB>", where AA, RR, GG and BB are hexadecimal digits.

Vertex

Required. Any expression that returns a Vertex value that describes this arc. The center point of the vertex is the center point of the arc. The two vertex handle points are used to find the start and end angle of the arc, by the angle each point makes with the center point.

The points do not describe absolute positions - that is controlled by the object's bounding box. What matters is the relation of the points to each other.

Comments:

This function is a layered graphics statement. For information about positioning a layered graphic, please refer to "Use Scaling to Position GUI Objects".

The Left and Right references are interchangeable. Whichever is smaller is taken as the left and the larger of the two values will be used as the right. The same is true of the top and bottom references. Note that the 1st 42 pixels of a VTScada application will be obscured by the title bar, if present.

Example:

```
press = GUIArc(0, 99, 99, 0,
               1, 1, 1, 1, 1 { Scaling },
               0, 0 { Movement },
               1, 0 { Visibility, Reserved },
               0, 0, 0 { Selectability },
               Pen("<FF000000>", 1, 1),
               Vertex(1,
                      Point(560, 512, Invalid, Invalid),
                      Point(623, 449, Invalid, Invalid),
                      Point(489, 583, Invalid, Invalid)));
```

This shows a black arc in the upper left corner of the window. No scaling is performed, and no animation is performed.
The first four parameters must be constants. See GUITransform for an example of how to compute the position dynamically.

Related Functions:

Arc | Circle | DrawArcPath | DrawChordPath | DrawEllipticalPath | DrawPiePath | Ellipse | GUIChord | GUIEllipse | GUIPie | GUITransform | NextFocusID | Normalize | Pie Point | Rotate | Trajectory | Vertex | VStatus

Related Information:

See: "Best Practices for Graphics" in the VTScada Programmer's Guide.

GUIBitmap

Description:	Draws an image of any of the following formats in a window. Can return a Boolean TRUE when selected by a mouse button or when the <ENTER> key is pressed after focus has been acquired. Available formats include, BMP, EMF, WMF, APM, CUT, PCX, JPG, PNG, and TIF
Returns:	GUI Object Return Codes
Usage:	Steady State only.
Function Groups:	Graphics
Format:	GUIBitmap(LeftReference, BottomReference, RightReference, TopReference, ScaleLeft, ScaleBottom, ScaleRight, ScaleTop, ScaleWhole, Trajectory, Rotation, Visibility, Reserved, Button, FocusID, FocusTrigger, Image)

Parameters:

LeftReference

A constant number that gives the left side reference coordinate. It must be a constant. A variable or expression is not valid here.

BottomReference

A constant number that gives the bottom side reference coordinate. It must be a constant. A variable or expression is not valid here. The top and bottom references are measured down from the top of the screen.

RightReference

A constant number that gives the right side reference coordinate. It must be a constant. A variable or expression is not valid here.

TopReference

A constant number that gives the top side reference coordinate. It must be a constant. A variable or expression is not valid here.

ScaleLeft

Required. Either a numeric expression, or any expression that returns a Normalize value. This parameter scales this side from its reference position with respect to the opposite side. If it is a numeric expression, a value of 1 will place the side at its reference position. A value of 0 will place it at the opposite side reference position. Similarly, a Normalize value will scale the side between the high and low limits. If the value is at the high level, the side will be at its reference position. If the value is at the low level, the side will be at the opposite side reference position.

ScaleBottom

Required. Either a numeric expression, or any expression that returns a Normalize value. This parameter scales this side from its reference position with respect to the opposite side. If it is a numeric expression, a value of 1 will place the side at its reference position. A value of 0 will place it at the opposite side reference position. Similarly, a Normalize value will scale the side between the high and low limits. If the value is at the high level, the side will be at its reference position. If the value is at the low level, the side will be at the opposite side reference position.

ScaleRight

Required. Either a numeric expression, or any expression that returns a Normalize value. This parameter scales this side from its reference position with respect to the opposite side. If it is a numeric expression, a value of 1 will place the side at its reference position. A value of 0 will place it at the opposite side reference position. Similarly, a Normalize value will scale the side between the high and low limits. If the value is at the high level, the side will be at its reference position. If the value is at the low level, the side will be at the opposite side reference position.

ScaleTop

Required. Either a numeric expression, or any expression that returns a Normalize value. This parameter scales this side from its reference position with respect to the opposite side. If it is a numeric expression, a value of 1 will place the side at its reference position. A value of 0 will place it at the opposite side reference position. Similarly, a Normalize value will scale the side between the high and low limits. If the value is at the high level, the side will be at its reference position. If the value is at the low level, the side will be at the

opposite side reference position.

ScaleWhole

Required. Either a numeric expression, or any expression that returns a Normalize value. This parameter scales the horizontal and vertical dimensions by the specified factor before the left, bottom, right and top coordinates are scaled.

Trajectory

Required. Either a Trajectory function, a variable containing a Trajectory value, or a numeric expression. If this is a Trajectory value or function, the appropriate translation is applied to the image after the rotation is applied. If it is a valid numeric expression, the image isn't translated, but is displayed. Any other value is Invalid.

Rotation

Required. Either a Rotate function, a variable containing a Rotate value, or a numeric expression. If this is a Rotate value or function, the appropriate rotation is applied to the image before the trajectory is applied. If it is a valid numeric expression, the image is rotated clockwise the number of degrees specified. Any other value is Invalid.

Visibility

Required. Any logical expression. If true, the image is drawn normally. If false, the image is not drawn.

Reserved

Reserved for future use, set to 0.

Button

Required. Any numeric expression giving the button combination that activates this graphic.

Value	Locator Buttons
0	No button combination will activate this graphic
1	Right button
2	Middle button
3	Right and middle buttons
4	Left button
5	Left and right buttons
6	Left and middle buttons
7	All three buttons

If the above values are multiplied by 8, the meaning for multiple buttons pressed becomes "OR" rather than "AND." For example, to accept any button on a 2 or 3 button mouse, use 56 (i.e. 8 * 7). To accept the left mouse button regardless of whether or not the right button is pressed, use 32 (i.e. 8 * 4).

If a 64 is added to this parameter, the function will become true when the mouse buttons are released rather than when they are pressed.

FocusID

Required. Any numeric expression giving the focus number of this graphic. If FocusID is 0, this graphic cannot receive the input focus. This parameter's value may be used in a NextFocusID statement to force this graphic to get the focus.

FocusTrigger

Required. Any Boolean expression. If FocusTrigger

changes from a valid false to a valid true, this graphic will acquire focus.

Image

Required. Either an image expression or a text expression. An image value will display that image. A text value identifies the name of an image file.
If an expression is used, this graphic can function like an Image Change, switching images in response to changing application states.

Comments:

32-bit color is used for all VTScada image drawing.
The performance of image drawing is heavily influenced by the presence or absence of graphics acceleration hardware. An anti-aliased image draws much more slowly than one without this option set, and becomes much less compatible with masking operations such as zColorChange.

This function is a layered graphics statement. See "Use Scaling to Position GUI Objects" for information about positioning a layered graphic.
The Left and Right references are interchangeable. Whichever is smaller is taken as the left and the larger of the two values will be used as the right. The same is true of the top and bottom references. Note that the 1st 42 pixels of a VTScada application will be obscured by the title bar, if present.

Example:

```
[
  { variable declaration & assignment }
  Left = 10; { scaling parameters }
  Right = 110;
  Top = 50;
  Bottom = 150;
]
{ ... main state for page or window }
GUIBitmap(0, 1, 1, 0 { Bounding box of image },
          1 - Top, Left, Bottom, 1 - Right, 1 { Scaling },
```

```
        0, 0 { No trajectory or rotation },
        1 { Bitmap is visible },
        0 { Reserved },
        0 { Left mouse button activates },
        0 { Focus ID number },
        FALSE { Focus trigger },
        "..\Bitmaps\Smiley face icon.bmp" { Bitmap file name });
```

This shows an image in the upper left corner of the window. Position and size are set using scaling parameters.

The first four parameters must be constants. See GUITransform for an example of how to compute the position dynamically.

Related Functions:

BitmapInfo | Crop | GUIButton | GUITransform | ImageArray | ImageSweep | MakeBitmap | ModifyBitmap | NextFocusID | Normalize | Rotate | Trajectory | VStatus

Related Information:

See: "Best Practices for Graphics" in the VTScada Programmer's Guide

GUIButton

Description: Draws a push-button in a window. Can return a Boolean TRUE when selected by a mouse button or when the <ENTER> key is pressed after focus has been acquired.

Returns: GUI Object Return Codes

Usage: Steady State only.

Function Groups: Graphics

Format: GUIButton(LeftReference, BottomReference, RightReference, TopReference, ScaleLeft, ScaleBottom, ScaleRight, ScaleTop, ScaleWhole, Trajectory, Rotation, Visibility, Reserved, Button, FocusID, FocusTrigger, Brush, HighlightPen, ShadowPen, TextColor, Sides, Reserved, UpLabel, DownLabel, Font, DownValue, UpValue, Variable[, ImageSet, ClickSound])

Parameters:

LeftReference

A constant number that gives the left side reference coordinate. It must be a constant. A variable or expression is not valid here.

BottomReference

A constant number that gives the bottom side reference coordinate. It must be a constant. A variable or expression is not valid here. The top and bottom references are measured down from the top of the screen.

RightReference

A constant number that gives the right side reference coordinate. It must be a constant. A variable or expression is not valid here.

TopReference

A constant number that gives the top side reference coordinate. It must be a constant. A variable or expression is not valid here.

ScaleLeft

Required. Either a numeric expression, or any expression that returns a Normalize value. This parameter scales this side from its reference position with respect to the opposite side. If it is a numeric expression, a value of 1 will place the side at its reference position. A value of 0 will place it at the opposite side reference position. Similarly, a Normalize value will scale the side between the high and low limits. If the value is at the high level, the side will be at its reference position. If the value is at the low level, the side will be at the opposite side reference position.

ScaleBottom

Required. Either a numeric expression, or any expression that returns a Normalize value. This parameter

scales this side from its reference position with respect to the opposite side. If it is a numeric expression, a value of 1 will place the side at its reference position. A value of 0 will place it at the opposite side reference position. Similarly, a Normalize value will scale the side between the high and low limits. If the value is at the high level, the side will be at its reference position. If the value is at the low level, the side will be at the opposite side reference position.

ScaleRight

Required. Either a numeric expression, or any expression that returns a Normalize value. This parameter scales this side from its reference position with respect to the opposite side. If it is a numeric expression, a value of 1 will place the side at its reference position. A value of 0 will place it at the opposite side reference position. Similarly, a Normalize value will scale the side between the high and low limits. If the value is at the high level, the side will be at its reference position. If the value is at the low level, the side will be at the opposite side reference position.

ScaleTop

Required. Either a numeric expression, or any expression that returns a Normalize value. This parameter scales this side from its reference position with respect to the opposite side. If it is a numeric expression, a value of 1 will place the side at its reference position. A value of 0 will place it at the opposite side reference position. Similarly, a Normalize value will scale the side between the high and low limits. If the value is at the high level, the side will be at its reference position. If the value is at the low level, the side will be at the opposite side reference position.

ScaleWhole

Required. Either a numeric expression, or any expression that returns a Normalize value. This parameter scales the horizontal and vertical dimensions by the specified factor before the left, bottom, right and top coordinates are scaled.

Trajectory

Required. Either a Trajectory function, a variable containing a Trajectory value, or a numeric expression. If this is a Trajectory value or function, the appropriate translation is applied to the image after the rotation is applied. If it is a valid numeric expression, the image isn't translated, but is displayed. Any other value is Invalid.

Rotation

Required. Either a Rotate function, a variable containing a Rotate value, or a numeric expression. If this is a Rotate value or function, the appropriate rotation is applied to the image before the trajectory is applied. If it is a valid numeric expression, the image is rotated clockwise the number of degrees specified. Any other value is Invalid.

Visibility

Required. Any logical expression. If true, the image is drawn normally. If false, the image is not drawn.

Reserved n/a

Reserved for future use, set to 0.

Button

Required. Any numeric expression giving the button combination that activates this graphic.

Value	Locator Buttons
0	No button combination will activate this graphic
1	Right button
2	Middle button
3	Right and middle buttons
4	Left button
5	Left and right buttons
6	Left and middle buttons
7	All three buttons

If the above values are multiplied by 8, the meaning for multiple buttons pressed becomes "OR" rather than "AND." For example, to accept any button on a 2 or 3 button mouse, use 56 (i.e. 8 * 7). To accept the left mouse button regardless of whether or not the right button is pressed, use 32 (i.e. 8 * 4).

If a 64 is added to this parameter, the function will become true when the mouse buttons are released rather than when they are pressed.

FocusID

Required. Any numeric expression giving the focus number of this graphic. If FocusID is zero, this graphic will not immediately have focus, but will work in any other manner. For values above and below zero, the control will be selectable until 32767, after which the graphic will not be visible.

This parameter's value may be used in a NextFocusID statement to force this graphic to get the focus.

FocusTrigger

Required. Any logical expression. If FocusTrigger changes from a valid false to a valid true, this graphic will attempt to obtain focus.

Brush

Required. Used to fill the background of the button. Any of the following may be used:

- a Brush object
- a palette index VTScada Color Palette
- a system color Constants for System Colors
- an RGB string with optional Alpha value in the format, "<AARRGGBB>", or "<RRGGBB>", where AA, RR, GG and BB are hexadecimal digits.

HighlightPen

Required. Used to draw the highlight sides of the button, one pixel wide. Any of the following may be used:

- a Pen object (in which case the pen's width value may be used)
- a palette index VTScada Color Palette
- a system color Constants for System Colors
- an RGB string with optional Alpha value in the format, "<AARRGGBB>", or "<RRGGBB>", where AA, RR, GG and BB are hexadecimal digits.

ShadowPen

Required. . Used to draw the shadowed sides of the button, one pixel wide. Any of the following may be used:

- a Pen object (in which case the pen's width value may be used)
- a palette index VTScada Color Palette

- a system color Constants for System Colors
- an RGB string with optional Alpha value in the format, "<AARRGGBB>", or "<RRGGBB>", where AA, RR, GG and BB are hexadecimal digits.

TextColor

Required.Used to set the color of the text that appears on the button. Any of the following may be used:

- a Pen object (only the color will be used)
- a palette index VTScada Color Palette
- a system color Constants for System Colors
- an RGB string with optional Alpha value in the format, "<AARRGGBB>", or "<RRGGBB>", where AA, RR, GG and BB are hexadecimal digits.

Sides

Required. Any numeric expression giving the number of sides on the button. If this value is 0, the button will have four sides. If not zero, the value must be greater than or equal to 4.
In general, the use of non-rectangular shapes is discouraged. When the button is selected, a rectangular outline will be shown.

Reserved

Reserved for future use, set to 0.

UpLabel

Required. An expression returning a Bitmap value, or text. If text, it will be drawn in the font set by the Font parameter. Both text and images will be centered on the button while it is up. Images will be scaled to fill the button area. To use an image file, use: MakeBitmap("relative\path\to\file") Any of the recognized image formats may be used; the file extension is not

required.

DownLabel

Required. An expression returning a Bitmap value, or text. If text, it will be drawn in the font set by the Font parameter, Both text and images will be centered on the button while it is down. Images will be scaled to fill the button area.

Font

Required. Any expression returning a Font value, or any numeric expression. Set as "0" to use the default system font.

DownValue

Required. Any expression. This will be assigned to the variable parameter when the button is down (pressed). If the value of Variable matches the value of this expression, the button will immediately be drawn as down.

UpValue

Required. Any expression. This will be assigned to the variable parameter when the button is up (released). If the value of Variable matches the value of this expression, the button will immediately be drawn as up.

Variable

Required. Any variable or any expression which may receive an assignment (any l-value). This will be set whenever the button is pressed or released. If this value is changed by another expression (or is given a default value) to match either the UpValue or the DownValue, the button will be drawn in whatever status the value represents - up or down.

ImageSet

A set of image values, stored in a structure, to

be used in response to various mouse actions. Not all need be specified.
The full list of possible members for the structure is as follows:

```
ImageSetStruct STRUCT [
  UpImage          { Image to use when the
button is "up" };
  DownImage       { Image to use when the but-
ton is "down" };
  MouseOverUpImage { Image to use when the
mouse is over an
                       "up" button };
  MouseOverDownImage { Image to use when the
mouse is over
                        a "down" button };
  DisabledImage    { Image to use when user
actions on
                        the button are dis-
abled };
]
```

ClickSound

The name of a .wav file that is to be played when the button is clicked. The extension ".wav" will be appended if not provided in the parameter. This file will not be played while another sound (for example, an alarm,) is sounding.

Comments: GUIButtons drawn on a page cannot be modified using ribbons or properties dialogs. Editing can only be done in code.
Standard practice is to use WinButtons if displaying plain text labels, and to use images for the labels when using a GUIButton.
There are three basic types of buttons:

- Momentary
- Toggle (or latching)
- Radio (which is actually a group of buttons behaving as a unit)

For details on how to create each of these types, see the Examples section.

A momentary button is one that immediately releases when pressed (i.e. it does not remain in its pressed position). It is usually used in conjunction with an If statement as an action trigger for a script.

A toggle or latching button is a button that when pressed remains locked (latched) into its pressed position until it is pressed again (unlatched). Its Variable parameter will be set according to the position of the button - if it is latched in (pressed), the variable will be set to 1, otherwise it will be 0. Similarly, if the variable's value is set by an external source from 0 to 1, the button will become pressed to match the value. This latching/toggling effect is achieved by the second last parameter being set to ! Variable, which is the same as saying "not the value of the Variable parameter" or more simply, "toggle the value from what it is".

A radio button is one of a set of latching buttons that are mutually exclusive, that is to say, only one button in the set may be latched (down) at one time. When another button in the same set is pressed, the current depressed button will "pop out". Two buttons cannot be latched in at once.

If the FocusID parameter of a GUIButton is to be used in a NextFocusID statement to force this graphic to get the focus, it is important to note that this does not cause the GUIButton statement to become true (selected), but only for it to become focused. Once the button is focused it may then be selected via the <CR> key on the keyboard.

Since structures cannot be initialized in steady state, the ImageSet parameter must be created in a script. You may create your own structures, or use one that has been created for you. The syntax to use the pre-defined structure varies depending on whether you are working in a script application or a standard VTScada application.

- For script applications, use \System\ImageSetStruct()
- For standard applications, use \ImageSetStruct().

The following set of rules defines the ImageSet behavior...

In the absence of the ImageSet parameter or, if that parameter is present but is not valid, the GUIButton statement will display a set of predefined renderings.

In the presence of a valid ImageSet parameter, the current GUIButton rendering will be disabled. (Note that providing a valid variable that holds Invalid for this parameter will not disable the older GUIButton rendering).

It is possible to specify the ImageSet parameter as Invalid and use only the ClickSound parameter

The ImageSet parameter, when valid, must consist of a structure holding the images.

Each structure member must be named as described in the parameter listing above.

The only valid formats for each image in the structure are image values.

You may declare your own structure or use one of the pre-created structures in VTScada. For standard apps you can simply use \ImageSetStruct ()and for script apps you can use \System\ImageSetStruct()

In addition to these rules for the ImageSet structure,

the following rules govern how the button will be rendered:

- If no image is provided in the set, no images shall be displayed for the button. You will end up with a blank, clickable area
- If the button is disabled, the "Disabled" image will be displayed at all times. If there is no Disabled image the "Up" image will be displayed.
- If the mouse cursor is not within the bounding box of the button:
- If the button state is logically up, the "Up" image (if any) shall be displayed. If there is no "Up" image, no image shall be displayed.
- If the button state is logically down, the "Down" image (if any) shall be displayed. If there is no "Down" image, no image shall be displayed.
- If the mouse cursor is within the bounding box of the button:
- If the button state is logically up, the "MouseOverUp" image (if any) shall be displayed. If there is no "MouseOverUp" image, the "Up" image (if any) shall be displayed. If there is no "Up" image, no image shall be displayed.
- If the button state is logically down, the "MouseOver-Down" image (if any) shall be displayed. If there is no "MouseOverDown" image, the "Down" image (if any) shall be displayed. If there is no "Down" image, no image shall be displayed.

The control has the ability to display text layered on top of the images. The text is displayed regardless of which image is being displayed. The text must be provided by the existing UpLabel, DownLabel, Tex-tColor and Font parameters. If the UpLabel and

DownLabel parameters are images, then they will not be used if the ImageSet parameter has prevented old GUIButton rendering.

The following parameters do not affect how the new images are displayed:

Brush, HighlightPen, ShadowPen, Sides.

This function is a layered graphics statement. See "Use Scaling to Position GUI Objects" for information about positioning a layered graphic.

The Left and Right references are interchangeable. Whichever is smaller is taken as the left and the larger of the two values will be used as the right. The same is true of the top and bottom references. Note that the 1st 42 pixels of a VTScada application will be obscured by the title bar, if present.

Example:

The following illustrates how to create a momentary button used as an action trigger for a script. Notice that in this particular example images are used to label the button rather than text. The image files are assumed to be in the Bitmaps folder, while the code for this module is in the Pages folder.

```
If GUIButton(10, 90, 100, 50 { Outline of the button },
             1, 1, 1, 1, 1 { No scaling },
             0, 0, 1, 0 { No movement; visible },
             64 + 4, 1, 0 { Triggered by left mouse
             button release },
             GetSystemColor(15){ Windows button face color },
             GetSystemColor(20){ Windows button highlight color},
             GetSystemColor(16){ Windows button shadow color },
             GetSystemColor(18){ Windows button text color },
             0, 0 { Windows standard attributes },
             MakeBitmap("..\Bitmaps\StartPict"),
             MakeBitmap("..\Bitmaps\StopPict") { Up and down labels
},
             0, 0, 1, 2 { No variable assignment });
[
  ...
]
```

This statement is a latching button that sets the value of the variable called pos:

```
GUIButton(10, 90, 100, 50 { Outline of the button },
          1, 1, 1, 1, 1 { No scaling },
          0, 0 { No movement },
          1, 0 { Visible; reserved },
          64 + 4, 1, 0 { Left mouse button release },
          7, 15, 8, 0 { Colors },
          0, 0 { Number of sides; reserved },
          MakeBitmap("..\Bitmaps\StartPict"),
          MakeBitmap("..\Bitmaps\StopPict") { Up and down labels },
          1, ! pos, pos { Latching });
```

To situate a button at a position given by left, bottom, right, top:

```
[ { variable declaration and initialization }
  left = 10;
  right = 100;
  top = 50;
  bottom = 140;
]

{ ... state code ... }
If GUIButton(0, 1, 1, 0 { Unit bounding box },
             1 - (left) { Left scaling }, bottom { Bottom scaling },
             right { Right scaling },1 - (top) { Top scaling },
             1, 0, 0 { No overall scaling/movement },
             1, 0, 68, 2, 0 { Visible; selectable },
             7, 15, 8, 0, 4, 0 { Colors; sides },
             MakeBitmap("..\Bitmaps\StartPict"),
             MakeBitmap("..\Bitmaps\StopPict") { Up and down labels
},
             0, 0, 1, 2 { No variable assignment });
[
  ...
]
```

To use the ImageSetStruct structure, you can either set the images in one statement like so: (note that in a script application, \System must be pre-fixed. These file names assume that the files are in the same folder as the module.)

```
ButtonImages = \ImageSetStruct(MakeBitMap("UpButton.png"),
                               MakeBitmap("DownButton.png"),
                               MakeBitmap("HoverUpButton.png"),
                               MakeBitmap("HoverDownButton.png"),
                               MakeBitmap("DisabledButton.png");
```

You can also set the images in multiple statements (which is recommended as better practice):

```
ButtonImages                   = \ImageSetStruct();
ButtonImages\UpImage            = MakeBitMap("UpButton.png");
```

```
ButtonImages\MouseOverUpImage   = MakeBitmap("HoverUpButton.png");
ButtonImages\DownImage          = MakeBitmap("DownButton.png");
ButtonImages\MouseOverDownImage = MakeBitmap("HoverDownButton.png");
ButtonImages\DisabledImage      = MakeBitmap("DisabledButton.png");
```

Having initialized the structure as shown, you can then use it in a GUIButton:

```
GUIButton(10, 90, 100, 50    { Outline of the button },
          1, 1, 1, 1, 1      { No scaling },
          0, 0, 1, 0         { No movement; visible },
          64 + 4, 1, 0       { Left mouse button release },
          GetSystemColor(15){ Windows button face color },
          GetSystemColor(20){ Windows button highlight color},
          GetSystemColor(16){ Windows button shadow color },
          GetSystemColor(18){ Windows button text color },
          0, 0               { Windows standard attributes },
          "Up", "Down" { Up and down labels },
          0, 0, 1, 2         { No variable assignment },
          ButtonImages, "Click.WAV");
```

Related Functions:

BitmapInfo | Crop | Brush | GUIBitmap | ImageArray | ImageSweep | MakeBitmap | ModifyBitmap | NextFocusID | Normalize | Pen | Rotate | Trajectory | VStatus | ZButton

Related Information:

See: "Best Practices for Graphics" in the VTScada Programmer's Guide

GUIChord

Description:	Draws a chord in a window. Can return a Boolean TRUE when selected by a mouse button or when the <ENTER> key is pressed after focus has been acquired.
Returns:	GUI Object Return Codes
Usage:	Steady State only.
Function Groups:	Graphics
Format:	GUIChord(LeftReference, BottomReference, RightReference, TopReference, ScaleLeft, ScaleBottom, ScaleRight, ScaleTop, ScaleWhole, Trajectory, Rotation, Opacity, Reserved, Button, FocusID, FocusTrigger, Brush, Pen, Vertex)

Parameters:

LeftReference

A constant number that gives the left side reference coordinate. It must be a constant. A variable or expression is not valid here.

BottomReference

A constant number that gives the bottom side reference coordinate. It must be a constant. A variable or expression is not valid here. The top and bottom references are measured down from the top of the screen.

RightReference

A constant number that gives the right side reference coordinate. It must be a constant. A variable or expression is not valid here.

TopReference

A constant number that gives the top side reference coordinate. It must be a constant. A variable or expression is not valid here.

ScaleLeft

Required. Either a numeric expression, or any expression that returns a Normalize value. This parameter scales this side from its reference position with respect to the opposite side. If it is a numeric expression, a value of 1 will place the side at its reference position. A value of 0 will place it at the opposite side reference position. Similarly, a Normalize value will scale the side between the high and low limits. If the value is at the high level, the side will be at its reference position. If the value is at the low level, the side will be at the opposite side reference position.

ScaleBottom

Required. Either a numeric expression, or any expression that returns a Normalize value. This parameter scales this side from its reference position with respect to the opposite side. If it is a numeric expression, a value of 1 will place the side at its reference position. A value of 0 will place it at the opposite side reference position. Similarly, a Normalize value will scale the side between the high and low limits. If the value is at the high level, the side will be at its reference position. If the value is at the low level, the side will be at the opposite side reference position.

ScaleRight

Required. Either a numeric expression, or any expression that returns a Normalize value. This parameter scales this side from its reference position with respect to the opposite side. If it is a numeric expression, a value of 1 will place the side at its reference position. A value of 0 will place it at the opposite side reference position. Similarly, a Normalize value will scale the side between the high and low limits. If the value is at the high level, the side will be at its reference position. If the value is at the low level, the side will be at the opposite side reference position.

ScaleTop

Required. Either a numeric expression, or any expression that returns a Normalize value. This parameter scales this side from its reference position with respect to the opposite side. If it is a numeric expression, a value of 1 will place the side at its reference position. A value of 0 will place it at the opposite side reference position. Similarly, a Normalize value will scale the side between the high and low limits. If the value is at the high level, the side will be at its reference position. If the value is at the low level, the side will be at the

opposite side reference position.

ScaleWhole

Required. Either a numeric expression, or any expression that returns a Normalize value. This parameter scales the horizontal and vertical dimensions by the specified factor before the left, bottom, right and top coordinates are scaled.

Trajectory

Required. Either a Trajectory function, a variable containing a Trajectory value, or a numeric expression. If this is a Trajectory value or function, the appropriate translation is applied to the image after the rotation is applied. If it is a valid numeric expression, the image isn't translated, but is displayed. Any other value is Invalid.

Rotation

Required. Either a Rotate function, a variable containing a Rotate value, or a numeric expression. If this is a Rotate value or function, the appropriate rotation is applied to the image before the trajectory is applied. If it is a valid numeric expression, the image is rotated clockwise the number of degrees specified. Any other value is Invalid.

Opacity

Required. Any Numeric expression, setting the opacity of the object. A value of one results in a solid, zero is invisible and values between zero and one are used as an alpha setting for opacity.

Reserved n/a

Reserved for future use, set to 0.

Button

Required. Any numeric expression giving the button combination that activates this graphic.

Value	Locator Buttons
0	No buttons
1	Right button
2	Middle button
3	Right and middle buttons
4	Left button
5	Left and right buttons
6	Left and middle buttons
7	All three buttons

If the above values are multiplied by 8, the meaning for multiple buttons pressed becomes "OR" rather than "AND." For example, to accept any button on a 2 or 3 button mouse, use 56 (i.e. 8 * 7). To accept the left mouse button regardless of whether or not the right button is pressed, use 32 (i.e. 8 * 4).

If a 64 is added to this parameter, the function will become true when the mouse buttons are released rather than when they are pressed.

FocusID

Required. Any numeric expression giving the focus number of this graphic. If FocusID is zero, this graphic cannot receive the input focus. This parameter's value may be used in a NextFocusID statement to force this graphic to get the focus.

FocusTrigger

Required. Any logical expression. If FocusTrigger changes from a valid false to a valid true, this graphic

will attempt to obtain focus.

Brush

Required. Any expression that returns a Brush value to be used to describe the fill. For a solid color fill, you may use any of the following:

- a palette index VTScada Color Palette
- a system color Constants for System Colors
- -1 (transparent)
- an RGB string with optional Alpha value in the format, "<AARRGGBB>", or "<RRGGBB>", where AA, RR, GG and BB are hexadecimal digits.

Pen

Required. Any expression that returns a Pen value to describe the color, width and line style of the border. For a 1-pixel solid border, you may use any of the following:

- a palette index VTScada Color Palette
- a system color Constants for System Colors
- -1 (transparent)
- an RGB string with optional Alpha value in the format, "<AARRGGBB>", or "<RRGGBB>", where AA, RR, GG and BB are hexadecimal digits.

Vertex

Required. Any expression that returns a Vertex value that describes this chord. The center point of the vertex is the center point of the ellipse that describes the chord. The two vertex handle points are used to find the start and end angle of the chord, by the angle each point makes with the center point.

Comments: This function is a layered graphics statement. See "Use Scaling to Position GUI Objects" for information about positioning a layered graphic.
The Left and Right references are interchangeable.

Whichever is smaller is taken as the left and the larger of the two values will be used as the right. The same is true of the top and bottom references. Note that the 1st 42 pixels of a VTScada application will be obscured by the title bar, if present.

Example:

```
GUIChord(400, 150, 600, 55 { Bounding box of the chord },
         1, 1, 1, 1, 1 { No scaling },
         0, 0 { No trajectory or rotation },
         1 { Chord is visible },
         0 { Reserved },
         0 { Not activated },
         6 { Focus ID number },
         FALSE { No focus trigger },
         Brush(12, 0, 0) { Brush light red, background
               ignored (style is solid) },
         Pen(15, 1, 1) { Pen draws white solid lines 1 pixel wide },
         Vertex(0 { Rectangular mode },
            Point(50, 50, Invalid, Invalid){ Center },
            Point(50, 0, Invalid, Invalid) { Start angle },
            Point(0, 50, Invalid, Invalid) { End angle }));
```

This shows a chord in the upper right corner of the window. If it is selected with the left mouse button, on the release of the mouse button the statement will return true. This is also the case if it is focused and the return key is pressed. This chord will attempt to get the keyboard input focus when the "F" key is pressed (because of the MatchKeys function in the trigger). It is focus number 6. This is the number to use in the NextFocusID function to force this graphic to get the focus. No scaling is performed, and no animation is performed.
The first four parameters must be constants. See GUITransform for an example of how to compute the position dynamically.

Related Functions:

DrawArcPath | DrawChordPath | DrawEllipticalPath | DrawPiePath | Ellipse GUIArc | GUIEllipse | GUIPie | GUITransform | NextFocusID | Normalize | Pie Point | Rotate | Trajectory | Vertex | VStatus

Related Information:

See: "Best Practices for Graphics" in the VTScada Programmer's Guide

GUIEllipse

Description: Draws an ellipse in a window. Can return a Boolean TRUE when selected by a mouse button or when the <ENTER> key is pressed after focus has been acquired.

Returns: GUI Object Return Codes

Usage: Steady State only.

Function Groups: Graphics

Format: GUIEllipse(LeftReference, BottomReference, RightReference, TopReference, ScaleLeft, ScaleBottom, ScaleRight, ScaleTop, ScaleWhole, Trajectory, Rotation, Opacity, Reserved, Button, FocusID, FocusTrigger, Brush, Pen)

Parameters:

LeftReference

A constant number that gives the left side reference coordinate. It must be a constant. A variable or expression is not valid here.

BottomReference

A constant number that gives the bottom side reference coordinate. It must be a constant. A variable or expression is not valid here. The top and bottom references are measured down from the top of the screen.

RightReference

A constant number that gives the right side reference coordinate. It must be a constant. A variable or expression is not valid here.

TopReference

A constant number that gives the top side reference coordinate. It must be a constant. A variable or expression is not valid here.

ScaleLeft

Required. Either a numeric expression, or any expression that returns a Normalize value. This parameter scales this side from its reference position with respect to the opposite side. If it is a numeric expression, a value of 1 will place the side at its reference position. A value of 0 will place it at the opposite side reference position. Similarly, a Normalize value will scale the side between the high and low limits. If the value is at the high level, the side will be at its reference position. If the value is at the low level, the side will be at the opposite side reference position.

ScaleBottom

Required. Either a numeric expression, or any expression that returns a Normalize value. This parameter scales this side from its reference position with respect to the opposite side. If it is a numeric expression, a value of 1 will place the side at its reference position. A value of 0 will place it at the opposite side reference position. Similarly, a Normalize value will scale the side between the high and low limits. If the value is at the high level, the side will be at its reference position. If the value is at the low level, the side will be at the opposite side reference position.

ScaleRight

Required. Either a numeric expression, or any expression that returns a Normalize value. This parameter scales this side from its reference position with respect to the opposite side. If it is a numeric expression, a value of 1 will place the side at its reference position. A value of 0 will place it at the opposite side reference position. Similarly, a Normalize value will scale the side between the high and low limits. If the value is at the high level, the side will be at its reference position.

If the value is at the low level, the side will be at the opposite side reference position.

ScaleTop

Required. Either a numeric expression, or any expression that returns a Normalize value. This parameter scales this side from its reference position with respect to the opposite side. If it is a numeric expression, a value of 1 will place the side at its reference position. A value of 0 will place it at the opposite side reference position. Similarly, a Normalize value will scale the side between the high and low limits. If the value is at the high level, the side will be at its reference position. If the value is at the low level, the side will be at the opposite side reference position.

ScaleWhole

Required. Either a numeric expression, or any expression that returns a Normalize value. This parameter scales the horizontal and vertical dimensions by the specified factor before the left, bottom, right and top coordinates are scaled.

Trajectory

Required. Either a Trajectory function, a variable containing a Trajectory value, or a numeric expression. If this is a Trajectory value or function, the appropriate translation is applied to the image after the rotation is applied. If it is a valid numeric expression, the image isn't translated, but is displayed. Any other value is Invalid.

Rotation

Required. Either a Rotate function, a variable containing a Rotate value, or a numeric expression. If this is a Rotate value or function, the appropriate rotation is applied to the image before the trajectory is applied. If it is a valid numeric expression, the image is rotated

clockwise the number of degrees specified. Any other value is Invalid.

Opacity

Required. Any Numeric expression, setting the opacity of the object. A value of one results in a solid, zero is invisible and values between zero and one are used as an alpha setting for opacity.

Reserved n/a

Reserved for future use, set to 0.

Button

Required. Any numeric expression giving the button combination that activates this graphic.

Value	Locator Buttons
0	No buttons
1	Right button
2	Middle button
3	Right and middle buttons
4	Left button
5	Left and right buttons
6	Left and middle buttons
7	All three buttons

If the above values are multiplied by 8, the meaning for multiple buttons pressed becomes "OR" rather than "AND." For example, to accept any button on a 2 or 3 button mouse, use 56 (i.e. 8 * 7). To accept the left mouse button regardless of whether or not the right button is pressed, use 32 (i.e. 8 * 4).

If a 64 is added to this parameter, the function will become true when the mouse buttons are

released rather than when they are pressed.

FocusID

Required. Any numeric expression giving the focus number of this graphic. If FocusID is zero, this graphic cannot receive the input focus. This parameter's value may be used in a NextFocusID statement to force this graphic to get the focus.

FocusTrigger

Required. Any logical expression. If FocusTrigger changes from a valid false to a valid true, this graphic will attempt to obtain focus.

Brush

Required. Any expression that returns a Brush value to be used to describe the fill. For a solid color fill, you may use any of the following:

- a palette index VTScada Color Palette
- a system color Constants for System Colors
- -1 (transparent)
- an RGB string with optional Alpha value in the format, "<AARRGGBB>", or "<RRGGBB>", where AA, RR, GG and BB are hexadecimal digits.

Pen

Required. Any expression that returns a Pen value to describe the color, width and line style of the border. For a 1-pixel solid border, you may use any of the following:

- a palette index VTScada Color Palette
- a system color Constants for System Colors
- -1 (transparent)
- an RGB string with optional Alpha value in the format, "<AARRGGBB>", or "<RRGGBB>", where AA, RR, GG and BB are hexadecimal digits.

Comments: This function is a layered graphics statement. See "Use Scaling to Position GUI Objects" for information about positioning a layered graphic.
GUIEllipses whose Pen value has an even numbered width are subject to pixel rounding errors, which is particularly obvious when drawing two concentric ellipses, one with an even width outline and one with an odd. To avoid this, always use all even or all odd outline widths for concentric ellipses.
The Left and Right references are interchangeable. Whichever is smaller is taken as the left and the larger of the two values will be used as the right. The same is true of the top and bottom references. Note that the 1st 42 pixels of a VTScada application will be obscured by the title bar, if present.

Example:

```
GUIEllipse(215, 128, 315, 58,
           1, 1, 1, 1, 1 { Scaling },
           0, 0 { Movement },
           1, 0 { Visibility, Reserved },
           0, 0, 0 { Selectability },
           Brush("<FFA0A0A0>", 0, 1), Pen("<FF000000>", 1, 1));
```

A basic gray ellipse with a single-pixel-wide, black border.
The first four parameters must be constants. See GUITransform for an example of how to compute the position dynamically.

Related Functions:

DrawArcPath | DrawChordPath | DrawEllipticalPath | DrawPiePath | GUIArc | GUIChord | GUIPie | GUITransform | NextFocusID | Normalize | Point | Rotate | Trajectory | Vertex | VStatus

Related Information:

See: "Best Practices for Graphics" in the VTScada Programmer's Guide

GUIPie

Description: Draws a pie-shaped wedge in a window. Can return a

Boolean TRUE when selected by a mouse button or when the <ENTER> key is pressed after focus has been acquired.

Returns: GUI Object Return Codes

Usage: Steady State only.

Function Groups: Graphics

Format: GUIPie(LeftReference, BottomReference, RightReference, TopReference, ScaleLeft, ScaleBottom, ScaleRight, ScaleTop, ScaleWhole, Trajectory, Rotation, Opacity, Reserved, Button, FocusID, FocusTrigger, Brush, Pen, Vertex)

Parameters:

LeftReference

A constant number that gives the left side reference coordinate. It must be a constant. A variable or expression is not valid here.

BottomReference

A constant number that gives the bottom side reference coordinate. It must be a constant. A variable or expression is not valid here. The top and bottom references are measured down from the top of the screen.

RightReference

A constant number that gives the right side reference coordinate. It must be a constant. A variable or expression is not valid here.

TopReference

A constant number that gives the top side reference coordinate. It must be a constant. A variable or expression is not valid here.

ScaleLeft

Required. Either a numeric expression, or any expression that returns a Normalize value. This parameter scales this side from its reference position with respect to the opposite side. If it is a numeric expression, a value of 1 will place the side at its reference position. A value of 0 will place it at the opposite side reference position. Similarly, a Normalize value will scale the side between the high and low limits. If the value is at the high level, the side will be at its reference position. If the value is at the low level, the side will be at the opposite side reference position.

ScaleBottom

Required. Either a numeric expression, or any expression that returns a Normalize value. This parameter scales this side from its reference position with respect to the opposite side. If it is a numeric expression, a value of 1 will place the side at its reference position. A value of 0 will place it at the opposite side reference position. Similarly, a Normalize value will scale the side between the high and low limits. If the value is at the high level, the side will be at its reference position. If the value is at the low level, the side will be at the opposite side reference position.

ScaleRight

Required. Either a numeric expression, or any expression that returns a Normalize value. This parameter scales this side from its reference position with respect to the opposite side. If it is a numeric expression, a value of 1 will place the side at its reference position. A value of 0 will place it at the opposite side reference position. Similarly, a Normalize value will scale the side between the high and low limits. If the value is at the high level, the side will be at its reference position. If the value is at the low level, the side will be at the

opposite side reference position.

ScaleTop

Required. Either a numeric expression, or any expression that returns a Normalize value. This parameter scales this side from its reference position with respect to the opposite side. If it is a numeric expression, a value of 1 will place the side at its reference position. A value of 0 will place it at the opposite side reference position. Similarly, a Normalize value will scale the side between the high and low limits. If the value is at the high level, the side will be at its reference position. If the value is at the low level, the side will be at the opposite side reference position.

ScaleWhole

Required. Either a numeric expression, or any expression that returns a Normalize value. This parameter scales the horizontal and vertical dimensions by the specified factor before the left, bottom, right and top coordinates are scaled.

Trajectory

Required. Either a Trajectory function, a variable containing a Trajectory value, or a numeric expression. If this is a Trajectory value or function, the appropriate translation is applied to the image after the rotation is applied. If it is a valid numeric expression, the image isn't translated, but is displayed. Any other value is Invalid.

Rotation

Required. Either a Rotate function, a variable containing a Rotate value, or a numeric expression. If this is a Rotate value or function, the appropriate rotation is applied to the image before the trajectory is applied. If it is a valid numeric expression, the image is rotated clockwise the number of degrees specified. Any other

value is Invalid.

Opacity

Required. Any Numeric expression, setting the opacity of the object. A value of one results in a solid, zero is invisible and values between zero and one are used as an alpha setting for opacity.

Reserved n/a

Reserved for future use, set to 0.

Button

Required. Any numeric expression giving the button combination that activates this graphic.

Value	Locator Buttons
0	No button combination will activate this graphic
1	Right button
2	Middle button
3	Right and middle buttons
4	Left button
5	Left and right buttons
6	Left and middle buttons
7	All three buttons

If the above values are multiplied by 8, the meaning for multiple buttons pressed becomes "OR" rather than "AND." For example, to accept any button on a 2 or 3 button mouse, use 56 (i.e. 8 * 7). To accept the left mouse button regardless of whether or not the right button is pressed, use 32 (i.e. 8 * 4).

If a 64 is added to this parameter, the function will become true when the mouse buttons are

released rather than when they are pressed.

FocusID

Required. Any numeric expression giving the focus number of this graphic. If FocusID is zero, this graphic cannot receive the input focus. This parameter's value may be used in a NextFocusID statement to force this graphic to get the focus.

FocusTrigger

Required. Any logical expression. If FocusTrigger changes from a valid false to a valid true, this graphic will attempt to obtain focus.

Brush

Required. Any expression that returns a Brush value to be used to describe the fill. For a solid color fill, you may use any of the following:

- a palette index VTScada Color Palette
- a system color Constants for System Colors
- -1 (transparent)
- an RGB string with optional Alpha value in the format, "<AARRGGBB>", or "<RRGGBB>", where AA, RR, GG and BB are hexadecimal digits.

Pen

Required. Any expression that returns a Pen value to describe the color, width and line style of the border. For a 1-pixel solid border, you may use any of the following:

- a palette index VTScada Color Palette
- a system color Constants for System Colors
- -1 (transparent)
- an RGB string with optional Alpha value in the format, "<AARRGGBB>", or "<RRGGBB>", where AA, RR, GG and BB are hexadecimal digits.

Vertex

Required. Any expression that returns the vertex that describes this pie. The center point of the vertex is the center point of the ellipse that describes the pie. The two vertex handle points are used to find the start and end angle of the pie, by the angle each point makes with the center point.
The points do not describe absolute positions – that is controlled by the object's bounding box. What matters is the relation of the points to each other.

Comments: This function is a layered graphics statement. See "Use Scaling to Position GUI Objects" for information about positioning a layered graphic.
The Left and Right references are interchangeable. Whichever is smaller is taken as the left and the larger of the two values will be used as the right. The same is true of the top and bottom references. Note that the 1st 42 pixels of a VTScada application will be obscured by the title bar, if present.

Example:

```
GUIPie(30, 570, 100, 420 { Bounding box of the pie },
               1, 1, 1, 1, 1 { No scaling },
               0, 0 { No trajectory or rotation },
               1, 0 { Pie is visible; reserved },
               0 { Not activated by selection },
               0 { Focus ID number },
               0 { No focus trigger },
               Brush(4, 6, 25) { Brush is dark red, brown
                              background, style is brick },
               Pen(2, 2, 2) { dark green dashed line, 2 pixels wide
},
               Vertex(0 { Rectangular mode },
                    Point(30, 495, Invalid, Invalid) { Pie point },
                    Point(90, 495, Invalid, Invalid) { Start },
                    Point(30, 0, Invalid, Invalid) { End }));
```

This shows a red and brown brick patterned pie outlined in a dark green (slightly thick) line in the lower left corner of the window. No scaling is performed, and no animation is performed.

The first four parameters must be constants. See GUITransform for an example of how to compute the position dynamically.

Related Functions:

DrawArcPath | DrawChordPath | DrawEllipticalPath | DrawPiePath | GUIArc | GUIChord | GUIEllipse | GUITransform | NextFocusID | Normalize | Point | Rotate | Trajectory | Vertex | VStatus

Related Information:

See: "Best Practices for Graphics" in the VTScada Programmer's Guide

GUIPipe

Note: Deprecated. Do not use in new code.

GUIPipes from older applications will still be drawn, but will be displayed as a wide line. GUIPipe statements are no longer generated by users drawing pipes in their applications.

Description: Draws a 3 dimensional, shaded pipe in a window and returns an indication when selected by a mouse button or the <ENTER> key.

Returns: GUI Object Return Codes

Usage: Steady State only.

Function Groups: Graphics

Format: GUIPipe(LeftReference, BottomReference, RightReference, TopReference, ScaleLeft, ScaleBottom, ScaleRight, ScaleTop, ScaleWhole, Trajectory, Rotation, Visibility, Reserved, Button, FocusID, FocusTrigger, LowIndex, HighIndex, PixelWidth, Path)

Parameters:

LeftReference

A constant number that gives the left side reference coordinate. It must be a constant. A variable or expression is not valid here.

BottomReference

A constant number that gives the bottom side reference coordinate. It must be a constant. A variable or expression is not valid here. The top and bottom references are measured down from the top of the screen.

RightReference

A constant number that gives the right side reference coordinate. It must be a constant. A variable or expression is not valid here.

TopReference

A constant number that gives the top side reference coordinate. It must be a constant. A variable or expression is not valid here.

ScaleLeft

Required. Either a numeric expression, or any expression that returns a Normalize value. This parameter scales this side from its reference position with respect to the opposite side. If it is a numeric expression, a value of 1 will place the side at its reference position. A value of 0 will place it at the opposite side reference position. Similarly, a Normalize value will scale the side between the high and low limits. If the value is at the high level, the side will be at its reference position. If the value is at the low level, the side will be at the opposite side reference position.

ScaleBottom

Required. Either a numeric expression, or any expression that returns a Normalize value. This parameter scales this side from its reference position with respect to the opposite side. If it is a numeric expression, a value of 1 will place the side at its reference position. A value of 0 will place it at the opposite side reference

position. Similarly, a Normalize value will scale the side between the high and low limits. If the value is at the high level, the side will be at its reference position. If the value is at the low level, the side will be at the opposite side reference position.

ScaleRight

Required. Either a numeric expression, or any expression that returns a Normalize value. This parameter scales this side from its reference position with respect to the opposite side. If it is a numeric expression, a value of 1 will place the side at its reference position. A value of 0 will place it at the opposite side reference position. Similarly, a Normalize value will scale the side between the high and low limits. If the value is at the high level, the side will be at its reference position. If the value is at the low level, the side will be at the opposite side reference position.

ScaleTop

Required. Either a numeric expression, or any expression that returns a Normalize value. This parameter scales this side from its reference position with respect to the opposite side. If it is a numeric expression, a value of 1 will place the side at its reference position. A value of 0 will place it at the opposite side reference position. Similarly, a Normalize value will scale the side between the high and low limits. If the value is at the high level, the side will be at its reference position. If the value is at the low level, the side will be at the opposite side reference position.

ScaleWhole

Required. Either a numeric expression, or any expression that returns a Normalize value. This parameter scales the horizontal and vertical dimensions by the specified factor before the left, bottom, right and top

coordinates are scaled.

Trajectory

Required. Either a Trajectory function, a variable containing a Trajectory value, or a numeric expression. If this is a Trajectory value or function, the appropriate translation is applied to the image after the rotation is applied. If it is a valid numeric expression, the image isn't translated, but is displayed. Any other value is Invalid.

Rotation

Required. Either a Rotate function, a variable containing a Rotate value, or a numeric expression. If this is a Rotate value or function, the appropriate rotation is applied to the image before the trajectory is applied. If it is a valid numeric expression, the image is rotated clockwise the number of degrees specified. Any other value is Invalid.

Visibility

Required. Any logical expression. If true, the image is drawn normally. If false, the image is not drawn.

Reserved n/a

Reserved for future use, set to 0.

Button

Required. Any numeric expression giving the button combination that activates this graphic.

Value	Locator Buttons
0	No buttons
1	Right button
2	Middle button
3	Right and middle buttons
4	Left button
5	Left and right buttons
6	Left and middle buttons
7	All three buttons

If the above values are multiplied by 8, the meaning for multiple buttons pressed becomes "OR" rather than "AND." For example, to accept any button on a 2 or 3 button mouse, use 56 (i.e. 8 * 7). To accept the left mouse button regardless of whether or not the right button is pressed, use 32 (i.e. 8 * 4).

If a 64 is added to this parameter, the function will become true when the mouse buttons are released rather than when they are pressed.

FocusID

Required. Any numeric expression giving the focus number of this graphic. If FocusID is zero, this graphic cannot receive the input focus. This parameter's value may be used in a NextFocusID statement to force this graphic to get the focus.

FocusTrigger

Required. Any logical expression. If FocusTrigger changes from a valid false to a valid true, this graphic

will attempt to obtain focus.

LowIndex

Required. Any numeric expression specifying the low index into the current color palette. It is used in conjunction with the next parameter to adjust the brightness and contrast of the shading.

HighIndex

Required. Any numeric expression specifying the high index into the current color palette. It is used in conjunction with the previous parameter to adjust the brightness and contrast of the shading.

PixelWidth

Required. Any numeric expression specifying the width of the shaded pipe in pixels and is subject to applicable scaling factors.

Path

Required. Any expression that returns a Path value that is used to draw the pipe. This defines the pipe's shape. The points do not describe absolute positions – that is controlled by the object's bounding box. What matters is the relation of the points to each other.

Comments:

This function is a layered graphics statement. See "Use Scaling to Position GUI Objects" for information about positioning a layered graphic.

The pipe will be drawn with a miter effect, such that pipe segments meet at 45-degree angles. For best results, use a vertex mode of 4, which preserves right angles, as shown in the example.

The Left and Right references are interchangeable. Whichever is smaller is taken as the left and the larger of the two values will be used as the right. The same is true of the top and bottom references. Note that the 1st 42 pixels of a VTScada application will be obscured by the title bar, if

present.

Example:

```
bottomLeft = Point(385, 145, Invalid, Invalid);
topLeft = Point(385, 25, Invalid, Invalid);
topRight = Point(645, 25, Invalid, Invalid);
bottomRight = Point(645, 145, Invalid, Invalid);
GUIPipe(385, 145, 645, 25 { Bounding box of pipe },
        1, 1, 1, 1, 1 { No scaling },
        0, 0 { No trajectory or rotation },
        1, 0 { Pipe is visible; reserved },
        0, 0, 0 { Cannot be focused/selected },
        176, 239 { Very dark gray to very light gray },
        24 { Width of pipe in pixels },
        Path(1 { Closed path },
        Vertex(4 { Manhattan[1] mode, right angles preserved },
        bottomLeft, bottomLeft, bottomLeft),
        Vertex(4 { Manhattan mode, right angles preserved },
        topLeft, topLeft, topLeft),
        Vertex(4 { Manhattan mode, right angles preserved },
        topRight, topRight, topRight),
        Vertex(4 { Manhattan mode, right angles preserved },
        bottomRight, bottomRight, bottomRight)));
```

This draws a pipe that follows an orthoganal path. The pipe is 24 pixels wide and is shaded from a dark to a light gray.
The first four parameters must be constants. See GUITransform for an example of how to compute the position dynamically.

Related Functions:

GUITransform | Normalize | NextFocusID | Path | Pipe | Point | Rotate| Trajectory | Vertex | VStatus | ZPipe | GUIPolygon | PathDraw

Related Information:

See: "Best Practices for Graphics" in the VTScada Programmer's Guide

GUIPolygon

Description: Draws a multi-sided polygon in a window. Can return a Boolean TRUE when selected by a mouse button or when the <ENTER> key is pressed after focus has been

---

[1]Meaning that all lines are horizontal or vertical. Inspired by a skyline of tall, rectangular buildings.

acquired.

Returns: Numeric

Usage: Steady State only.

Function Groups: Graphics

Format: GUIPolygon(LeftReference, BottomReference, RightReference, TopReference, ScaleLeft, ScaleBottom, ScaleRight, ScaleTop, ScaleWhole, Trajectory, Rotation, Opacity, Reserved, Button, FocusID, FocusTrigger, Brush, Pen, Path)

Parameters:

LeftReference

A constant number that gives the left side reference coordinate. It must be a constant. A variable or expression is not valid here.

BottomReference

A constant number that gives the bottom side reference coordinate. It must be a constant. A variable or expression is not valid here. The top and bottom references are measured down from the top of the screen.

RightReference

A constant number that gives the right side reference coordinate. It must be a constant. A variable or expression is not valid here.

TopReference

A constant number that gives the top side reference coordinate. It must be a constant. A variable or expression is not valid here.

ScaleLeft

Required. Either a numeric expression, or any expression that returns a Normalize value. This parameter scales this side from its reference position with respect

to the opposite side. If it is a numeric expression, a value of 1 will place the side at its reference position. A value of 0 will place it at the opposite side reference position. Similarly, a Normalize value will scale the side between the high and low limits. If the value is at the high level, the side will be at its reference position. If the value is at the low level, the side will be at the opposite side reference position.

ScaleBottom

Required. Either a numeric expression, or any expression that returns a Normalize value. This parameter scales this side from its reference position with respect to the opposite side. If it is a numeric expression, a value of 1 will place the side at its reference position. A value of 0 will place it at the opposite side reference position. Similarly, a Normalize value will scale the side between the high and low limits. If the value is at the high level, the side will be at its reference position. If the value is at the low level, the side will be at the opposite side reference position.

ScaleRight

Required. Either a numeric expression, or any expression that returns a Normalize value. This parameter scales this side from its reference position with respect to the opposite side. If it is a numeric expression, a value of 1 will place the side at its reference position. A value of 0 will place it at the opposite side reference position. Similarly, a Normalize value will scale the side between the high and low limits. If the value is at the high level, the side will be at its reference position. If the value is at the low level, the side will be at the opposite side reference position.

ScaleTop

Required. Either a numeric expression, or any expres-

sion that returns a Normalize value. This parameter scales this side from its reference position with respect to the opposite side. If it is a numeric expression, a value of 1 will place the side at its reference position. A value of 0 will place it at the opposite side reference position. Similarly, a Normalize value will scale the side between the high and low limits. If the value is at the high level, the side will be at its reference position. If the value is at the low level, the side will be at the opposite side reference position.

ScaleWhole

Required. Either a numeric expression, or any expression that returns a Normalize value. This parameter scales the horizontal and vertical dimensions by the specified factor before the left, bottom, right and top coordinates are scaled.

Trajectory

Required. Either a Trajectory function, a variable containing a Trajectory value, or a numeric expression. If this is a Trajectory value or function, the appropriate translation is applied to the image after the rotation is applied. If it is a valid numeric expression, the image isn't translated, but is displayed. Any other value is Invalid.

Rotation

Required. Either a Rotate function, a variable containing a Rotate value, or a numeric expression. If this is a Rotate value or function, the appropriate rotation is applied to the image before the trajectory is applied. If it is a valid numeric expression, the image is rotated clockwise the number of degrees specified. Any other value is Invalid.

Opacity

Required. Any Numeric expression, setting the opacity of the object. A value of one results in a solid, zero is invisible and values between zero and one are used as an alpha setting for opacity.

Reserved n/a

Reserved for future use, set to 0.

Button

Required. Any numeric expression giving the button combination that activates this graphic.

Value	Locator Buttons
0	No buttons
1	Right button
2	Middle button
3	Right and middle buttons
4	Left button
5	Left and right buttons
6	Left and middle buttons
7	All three buttons

If the above values are multiplied by 8, the meaning for multiple buttons pressed becomes "OR" rather than "AND." For example, to accept any button on a 2 or 3 button mouse, use 56 (i.e. 8 * 7). To accept the left mouse button regardless of whether or not the right button is pressed, use 32 (i.e. 8 * 4).

If a 64 is added to this parameter, the function will become true when the mouse buttons are released rather than when they are pressed.

FocusID

Required. Any numeric expression giving the focus

number of this graphic. If FocusID is zero, this graphic cannot receive the input focus. This parameter's value may be used in a NextFocusID statement to force this graphic to get the focus.

FocusTrigger

Required. Any logical expression. If FocusTrigger changes from a valid false to a valid true, this graphic will attempt to obtain focus.

Brush

Required, but used only if the polygon is closed. Any expression that returns a Brush value to be used to describe the fill. For a solid color fill, you may use any of the following:

- a palette index VTScada Color Palette
- a system color Constants for System Colors
- -1 (transparent)
- an RGB string with optional Alpha value in the format, "<AARRGGBB>", or "<RRGGBB>", where AA, RR, GG and BB are hexadecimal digits.

Pen

Required. Any expression that returns a Pen value to describe the color, width and line style of the border. For a 1-pixel solid border, you may use any of the following:

- a palette index VTScada Color Palette
- a system color Constants for System Colors
- -1 (transparent)
- an RGB string with optional Alpha value in the format, "<AARRGGBB>", or "<RRGGBB>", where AA, RR, GG and BB are hexadecimal digits.

Path

Required. Any expression that returns a Path value that

is used to draw the polygon. This defines the polygon's shape.

The points do not describe absolute positions – that is controlled by the object's bounding box. What matters is the relation of the points to each other.

Comments: This function is a layered graphics statement. See "Use Scaling to Position GUI Objects" for information about positioning a layered graphic.

The handle points on each vertex control the Bezier curve used to draw that segment of the polygon.

The Left and Right references are interchangeable. Whichever is smaller is taken as the left and the larger of the two values will be used as the right. The same is true of the top and bottom references. Note that the 1st 42 pixels of a VTScada application will be obscured by the title bar, if present.

Example:

```
[
  { variable declaration & assignment }
  Left = 50; { scaling parameters }
  Right = 210;
  Top = 550;
  Bottom = 10;
]

{ ... main state for page or window... }
GUIPolygon(0, 1, 1, 0 { Bounding box of image },
          1 - Top, Left, Bottom, 1 - Right, 1 { Scaling },
           0, 0 { No trajectory or rotation },
           1, 0 { Polygon is visible; reserved },
           0, 0, 0 { Cannot be focused/selected },
           Brush(14, 2, 13) { Brush is yellow, dark green
           background, style crosshatched },
           Pen(2, 1, 3) { dark green solid line, 3 pixels wide },
           Path(1 { Closed figure },
             Vertex(0 { Rectangular mode, no curvature },
             Point(0, 100, Invalid, Invalid) { Center },
             Point(0, 100, Invalid, Invalid) { In handle },
             Point(0, 100, Invalid, Invalid) { Out handle }),
           Vertex(0 { Rectangular mode, no curvature },
             Point(100, 100, Invalid, Invalid){ Center },
             Point(100, 100, Invalid, Invalid){ In handle },
             Point(100, 100, Invalid, Invalid){ Out handle }),
           Vertex(0 { Rectangular mode, no curvature },
```

```
            Point(100, 0, Invalid, Invalid) { Center },
            Point(100, 0, Invalid, Invalid) { In handle },
            Point(100, 0, Invalid, Invalid) { Out handle }}));
```

This shows a triangle at the top of the window. It has a yellow crosshatched pattern on a dark green background and is outlined by a thick dark green line. It cannot be focused. No animation is performed. The first four parameters must be constants. See GUITransform for an example of how to compute the position dynamically.

Related Functions:

DrawPath | NextFocusID | Normalize | Point | Rotate | Trajectory | Vertex | VStatus

Related Information:

See: "Best Practices for Graphics" in the VTScada Programmer's Guide

GUIRectangle

Description: Draws a rectangle in a window. Can return a Boolean TRUE when selected by a mouse button or when the <ENTER> key is pressed after focus has been acquired.

Returns: GUI Object Return Codes

Usage: Steady State only.

Function Groups: Graphics

Format: GUIRectangle(LeftReference, BottomReference, RightReference, TopReference, ScaleLeft, ScaleBottom, ScaleRight, ScaleTop, ScaleWhole, Trajectory, Rotation, Opacity, Reserved, Button, FocusID, FocusTrigger, Brush, Pen)

Parameters:

LeftReference

A constant number that gives the left side reference coordinate. It must be a constant. A variable or expression is not valid here.

BottomReference

A constant number that gives the bottom side ref-

erence coordinate. It must be a constant. A variable or expression is not valid here. The top and bottom references are measured down from the top of the screen.

RightReference

A constant number that gives the right side reference coordinate. It must be a constant. A variable or expression is not valid here.

TopReference

A constant number that gives the top side reference coordinate. It must be a constant. A variable or expression is not valid here.

ScaleLeft

Required. Either a numeric expression, or any expression that returns a Normalize value. This parameter scales this side from its reference position with respect to the opposite side. If it is a numeric expression, a value of 1 will place the side at its reference position. A value of 0 will place it at the opposite side reference position. Similarly, a Normalize value will scale the side between the high and low limits. If the value is at the high level, the side will be at its reference position. If the value is at the low level, the side will be at the opposite side reference position.

ScaleBottom

Required. Either a numeric expression, or any expression that returns a Normalize value. This parameter scales this side from its reference position with respect to the opposite side. If it is a numeric expression, a value of 1 will place the side at its reference position. A value of 0 will place it at the opposite side reference position. Similarly, a Normalize value will scale the side between the high and low limits. If the value is at the high level, the side will be at its reference position.

If the value is at the low level, the side will be at the opposite side reference position.

ScaleRight

Required. Either a numeric expression, or any expression that returns a Normalize value. This parameter scales this side from its reference position with respect to the opposite side. If it is a numeric expression, a value of 1 will place the side at its reference position. A value of 0 will place it at the opposite side reference position. Similarly, a Normalize value will scale the side between the high and low limits. If the value is at the high level, the side will be at its reference position. If the value is at the low level, the side will be at the opposite side reference position.

ScaleTop

Required. Either a numeric expression, or any expression that returns a Normalize value. This parameter scales this side from its reference position with respect to the opposite side. If it is a numeric expression, a value of 1 will place the side at its reference position. A value of 0 will place it at the opposite side reference position. Similarly, a Normalize value will scale the side between the high and low limits. If the value is at the high level, the side will be at its reference position. If the value is at the low level, the side will be at the opposite side reference position.

ScaleWhole

Required. Either a numeric expression, or any expression that returns a Normalize value. This parameter scales the horizontal and vertical dimensions by the specified factor before the left, bottom, right and top coordinates are scaled.

Trajectory

Required. Either a Trajectory function, a variable containing a Trajectory value, or a numeric expression. If this is a Trajectory value or function, the appropriate translation is applied to the image after the rotation is applied. If it is a valid numeric expression, the image isn't translated, but is displayed. Any other value is Invalid.

Rotation

Required. Either a Rotate function, a variable containing a Rotate value, or a numeric expression. If this is a Rotate value or function, the appropriate rotation is applied to the image before the trajectory is applied. If it is a valid numeric expression, the image is rotated clockwise the number of degrees specified. Any other value is Invalid.

Opacity

Required. Any Numeric expression, setting the opacity of the object. A value of one results in a solid, zero is invisible and values between zero and one are used as an alpha setting for opacity.

Reserved n/a

Reserved for future use, set to 0.

Button

Required. Any numeric expression giving the button combination that activates this graphic.

Value	Locator Buttons
0	No buttons
1	Right button
2	Middle button
3	Right and middle buttons
4	Left button
5	Left and right buttons
6	Left and middle buttons
7	All three buttons

If the above values are multiplied by 8, the meaning for multiple buttons pressed becomes "OR" rather than "AND." For example, to accept any button on a 2 or 3 button mouse, use 56 (i.e. 8 * 7). To accept the left mouse button regardless of whether or not the right button is pressed, use 32 (i.e. 8 * 4).

If a 64 is added to this parameter, the function will become true when the mouse buttons are released rather than when they are pressed.

FocusID

FocusID is any numeric expression giving the focus number of this graphic. If FocusID is zero, this graphic cannot receive the input focus. This parameter's value may be used in a NextFocusID statement to force this graphic to get the focus.

FocusTrigger

Required. Any logical expression. If FocusTrigger changes from a valid false to a valid true, this graphic

will attempt to obtain focus.

Brush

Required. Any expression that returns a Brush value to be used to describe the fill. For a solid color fill, you may use any of the following:

- a palette index VTScada Color Palette
- a system color Constants for System Colors
- -1 (transparent)
- an RGB string with optional Alpha value in the format, "<AARRGGBB>", or "<RRGGBB>", where AA, RR, GG and BB are hexadecimal digits.

Pen

Required. Any expression that returns a Pen value to describe the color, width and line style of the border. For a 1-pixel solid border, you may use any of the following:

- a palette index VTScada Color Palette
- a system color Constants for System Colors
- -1 (transparent)
- an RGB string with optional Alpha value in the format, "<AARRGGBB>", or "<RRGGBB>", where AA, RR, GG and BB are hexadecimal digits.

Comments:

This function is a layered graphics statement. See "Use Scaling to Position GUI Objects" for information about positioning a layered graphic.
The Left and Right references are interchangeable. Whichever is smaller is taken as the left and the larger of the two values will be used as the right. The same is true of the top and bottom references. Note that the 1st 42 pixels of a VTScada application will be obscured by the title bar, if present.

Examples:

```
GUIRectangle(19, 126, 143, 58,
             1, 1, 1, 1, 1 { Scaling },
             0, 0 { Movement },
             1, 0 { Visibility, Reserved },
             0, 0, 0 { Selectability },
             Brush("<FFA0A0A0>", 0, 1), Pen("<FF000000>", 1, 1));
```

A basic gray rectangle with a single-pixel-wide, black border.

```
GUIRectangle(25, 100, 225, 80 { Bounding box of rectangle },
             1, 1, 1, 1, 1 { No scaling },
           0, 0 { No trajectory or rotation },
           1, 0 { Rectangle is visible; reserved },
           4 { Left mouse button },
           5 { Focus ID number },
           MatchKeys(2, "T"){ Focus trigger },
           Brush(14, 0, 1) { Brush is yellow, background
                             ignored (style is solid) },
           Pen(2, 1, 1) { Pen is dark green solid
                          line, 1 pixel wide });
```

This shows a rectangle in the upper left corner of the window. If it is clicked with the left mouse button, or if it is focused and the return key is pressed, it will return true. This rectangle will attempt to get the keyboard input focus when the "T" key is pressed (because of the MatchKeys function in the trigger). It is focus number 5. This is the number to use in the NextFocusID function to force this graphic to get the focus. No scaling is performed, and no animation is performed.
The first four parameters must be constants. See GUITransform for an example of how to compute the position dynamically.

Related Functions:

Bar | Box | NextFocusID | Normalize | Rotate | Trajectory | ZBar | ZBox| GUITransform | VStatus

Related Information:

See: "Best Practices for Graphics" in the VTScada Programmer's Guide

GUIText

Description: Draws formatted text in a window. Can return a Boolean TRUE when selected by a mouse button or when the <ENTER> key is pressed after focus has been acquired.

Returns: GUI Object Return Codes

Usage: Steady State only.

Function Groups: Graphics

Format: GUIText(LeftReference, BottomReference, RightReference, TopReference, ScaleLeft, ScaleBottom, ScaleRight, ScaleTop, ScaleWhole, Trajectory, Rotation, Opacity, Options, Button, FocusID, FocusTrigger, Brush, Pen, Font, HCenter, VCenter, Format, V1, V2, ...)

Parameters:

LeftReference

A constant number that gives the left side reference coordinate. It must be a constant. A variable or expression is not valid here.

BottomReference

A constant number that gives the bottom side reference coordinate. It must be a constant. A variable or expression is not valid here. The top and bottom references are measured down from the top of the screen.

RightReference

A constant number that gives the right side reference coordinate. It must be a constant. A variable or expression is not valid here.

TopReference

A constant number that gives the top side reference coordinate. It must be a constant. A variable or expression is not valid here.

ScaleLeft

Required. Either a numeric expression, or any expression that returns a Normalize value. This parameter scales this side from its reference position with respect

to the opposite side. If it is a numeric expression, a value of 1 will place the side at its reference position. A value of 0 will place it at the opposite side reference position. Similarly, a Normalize value will scale the side between the high and low limits. If the value is at the high level, the side will be at its reference position. If the value is at the low level, the side will be at the opposite side reference position.

ScaleBottom

Required. Either a numeric expression, or any expression that returns a Normalize value. This parameter scales this side from its reference position with respect to the opposite side. If it is a numeric expression, a value of 1 will place the side at its reference position. A value of 0 will place it at the opposite side reference position. Similarly, a Normalize value will scale the side between the high and low limits. If the value is at the high level, the side will be at its reference position. If the value is at the low level, the side will be at the opposite side reference position.

ScaleRight

Required. Either a numeric expression, or any expression that returns a Normalize value. This parameter scales this side from its reference position with respect to the opposite side. If it is a numeric expression, a value of 1 will place the side at its reference position. A value of 0 will place it at the opposite side reference position. Similarly, a Normalize value will scale the side between the high and low limits. If the value is at the high level, the side will be at its reference position. If the value is at the low level, the side will be at the opposite side reference position.

ScaleTop

Required. Either a numeric expression, or any expres-

sion that returns a Normalize value. This parameter scales this side from its reference position with respect to the opposite side. If it is a numeric expression, a value of 1 will place the side at its reference position. A value of 0 will place it at the opposite side reference position. Similarly, a Normalize value will scale the side between the high and low limits. If the value is at the high level, the side will be at its reference position. If the value is at the low level, the side will be at the opposite side reference position.

ScaleWhole

Required. Either a numeric expression, or any expression that returns a Normalize value. This parameter scales the horizontal and vertical dimensions by the specified factor before the left, bottom, right and top coordinates are scaled.

Trajectory

Required. Either a Trajectory function, a variable containing a Trajectory value, or a numeric expression. If this is a Trajectory value or function, the appropriate translation is applied to the image after the rotation is applied. If it is a valid numeric expression, the image isn't translated, but is displayed. Any other value is Invalid.

Rotation

Required. Either a Rotate function, a variable containing a Rotate value, or a numeric expression. If this is a Rotate value or function, the appropriate rotation is applied to the image before the trajectory is applied. If it is a valid numeric expression, the image is rotated clockwise the number of degrees specified. Any other value is Invalid.

Opacity

Required. Any Numeric expression, setting the opacity of the object. A value of one results in a solid, zero is invisible and values between zero and one are used as an alpha setting for opacity.

Options

Required. Any numeric expression giving how a font behaves with regards to scaling, as follows:

Options	Font Behavior
0	Font not transformed
1	Font is transformed.
2	Font is transformed, but only by the top-most transform in the calling path.

Button

Required. Any numeric expression giving the button combination that activates this graphic.

Button	Locator Buttons
0	No buttons
1	Right button
2	Middle button
3	Right and middle buttons
4	Left button
5	Left and right buttons
6	Left and middle buttons
7	All three buttons

If the above values are multiplied by 8, the meaning for multiple buttons pressed becomes "OR" rather than "AND." For example, to accept any button on a 2 or 3 button mouse, use 56 (i.e. 8 * 7). To accept the left mouse button regardless

of whether or not the right button is pressed, use 32 (i.e. 8 * 4).
If a 64 is added to this parameter, the function will become true when the mouse buttons are released rather than when they are pressed.

FocusID

Required. Any numeric expression giving the focus number of this graphic. If FocusID is zero, this graphic cannot receive the input focus. This parameter's value may be used in a NextFocusID statement to force this graphic to get the focus.

FocusTrigger

Required. Any logical expression. If FocusTrigger changes from a valid false to a valid true, this graphic will attempt to obtain focus.

Brush

Required. Any expression that returns a Brush value to describe the background. A solid-color background may be defined by using any of the following:

- a palette index VTScada Color Palette
- a system color Constants for System Colors
- -1 (transparent)
- an RGB string with optional Alpha value in the format, "<AARRGGBB>", or "<RRGGBB>", where AA, RR, GG and BB are hexadecimal digits.

Pen

Required. Any expression that returns a Pen value. Only the color property of the pen will be used. You may also use any of the following:

- a palette index VTScada Color Palette
- a system color Constants for System Colors
- -1 (transparent)

- an RGB string with optional Alpha value in the format, "<AARRGGBB>", or "<RRGGBB>", where AA, RR, GG and BB are hexadecimal digits.

Font

Required. Any expression that returns a Font value used to display the text.

HCenter

Required. Any numeric expression that specifies the horizontal justification and clipping, as shown in the following table:

Value	H. Justification	Clipping
0	Left	Right
1	Left	Both
2	Left	Left
3	Center	Right
4	Center	Both
5	Center	Left
6	Right	Right
7	Right	Both
8	Right	Left

VCenter

Required. Any numeric expression that specifies the vertical justification and clipping, as shown in the following table:

Value	V. Justification	Clipping
0	Top	Bottom
1	Top	Both
2	Top	Top
3	Center	Bottom
4	Center	Both
5	Center	Top
6	Bottom	Bottom
7	Bottom	Both
8	Bottom	Top

Format

Required. Any text expression giving the format of how the values (Vn parameters) are to be written. This format is similar, but not identical, to the C language format string for the printf function, whereby each of the Vn parameters in the GUIText statement is assigned to a % format specification in the order in which each appears in the list. Note that like a standard text string, these format specifiers must also be enclosed by double quotes. If a format specification appears for which there are no remaining V parameters, the format specification characters themselves are output to the stream exactly as they appear in the Format. For the % format specifications, the following form applies (where the [ ] indicates optional elements):

%[-][+][SPACE][#][width][.precision]type
where:
% is mandatory.
+ causes the data to be left justified within the field. For binary types b and ASCII character types c, this option is ignored.
- causes positive numbers to be prefaced with a + sign (negative numbers are unaffected). This allows easy alignment of positive and negative numbers in a printed column of numbers. For binary types b and non-numerical types, this option is ignored.
SPACE represents the single space character, and is similar to the [+] option but places a single space rather than a plus sign in front of positive numbers (negative numbers are still unaffected). This allows alignment of a column of numbers without having to show all signs. For binary types b and non-numerical types, this option is ignored.
When used with the o , x , or X format, the # flag prefixes any nonzero output value with 0, 0x, or 0X, respectively..
width is a number that specifies the minimum number of characters to output. Numbers that require more characters than specified by the width value will be output in their entirety without truncation. If the number of characters in the number or string is less than width, blanks will be added to the left or right, depending upon whether the output is left or right justified (i.e. whether or not the [-] option has been specified) until the width is reached. For

binary types b and ASCII character types c, this option is ignored.

.precision has a different meaning for each of the type options as follows:

Integer types d, i, u, o, x and X – precision specifies the minimum number of digits to output. If the number contains fewer digits, leading zeroes will be added to the left of the number. If precision is 0, omitted, or if the decimal point appears without a number following it, the precision defaults to 1. The number is not truncated.

Floating point types e and E – precision specifies the number of digits after the decimal point. The last digit is rounded. The default precision in this case is 6. If the precision is 0 or if the decimal point appears without a number following it, no decimal point appears in the output.

Floating point type f – precision specifies the number of digits after the decimal point. The last digit is rounded. The default precision is 0. If the precision is explicitly 0, no decimal point is output. If a decimal point is output, at least one digit will be placed before the decimal point.

Floating point types g and G – precision specifies the maximum number of significant digits to be output. If no precision is specified, all significant digits are written.

String type s – precision specifies the maximum number of characters of the string to be output. If the string contains more characters than spe-

cified by the precision, the string is truncated and only the first characters are written. If the precision is not specified, all of the string characters are output.

ASCII character type c - this option is ignored.

Binary type b - this option is ignored.

type is a mandatory specification that must be one of those listed here. Note that the case of the letter is important. Specifying a character for the type which is not in this list will result in all the characters following the % up to that point to be output exactly as they appear in the Format string.

Type	Meaning
nb	Binary format, where n is a number indicating the type of value (following table)
c	Single ASCII character (byte)
d	Signed decimal integer
e	Signed exponential. Exponent key is "e"
E	Signed exponential. Exponent key is "E"
f	Signed floating point
g	e or f format, whichever is shorter
G	E or f format, whichever is shorter
i	Signed decimal integer
o	Unsigned octal integer
s	Text string
u	Unsigned decimal integer
x	Unsigned hex integer using "abcdef"
X	Unsigned hex integer using "ABCDEF"

Binary type b – For the format specification of %nb, where n specifies the type of number, n must be a single digit from one of the following choices:

n Value	Type
0	Byte
1	Short integer (2 bytes, low byte first)
2	Long integer (4 bytes, low bytes first)
3	IEEE single precision float (4 bytes)
5	IEEE double precision float (8 bytes)

Notice that the other options such as width and precision do not apply to the b type.

ASCII character type c – This type is not representative of a single character in a string, but rather, represents single byte ASCII characters. Input values (the Vn parameter to which this format specification applies) must be integers in the range of 0 to 255 in order for the output to be a valid ASCII equivalent character. Strings are not acceptable input values. Note that the %c format specifier behaves differently when used in an output statement such as GUIText than when used in an input statement, such as BuffRead.

Text string type s – Text in the string is written exactly as it appears, with two exceptions. First, since format specifications for the Vn parameters are indicated by a percentage sign, to include an actual percentage sign as part of the text string, precede it with a backslash character (i.e. \%). Also, since the backslash character is

used in this manner, as well as with special control characters such as form feed, to write a backslash as part of the text string, use two backslash characters (i.e. \\).

Note: Note: Omitting the %s format character for parameters that are to be displayed as strings can result in errors such as "Area\Place" being displayed as "AreaPlace".

Control characters - In order to encode certain control characters as part of the Format parameter, one of two methods may be used. The first is to use a backslash character followed by one of the single character codes listed here to produce the desired result (notice that the letters must be lower case):

Code	Meaning
\b	Backspace
\f	Form feed
\t	Horizontal tab
\v	Vertical tab

In addition to the predefined codes, an alternate form may be used :

\nnn

Where nnn is a three digit integer in the range of 0 to 255 specifying a certain ASCII character. If the number contains less than three digits, the leading spaces must be padded with zeroes. This is not the case with the previously listed single character control characters. For example, to include the one byte ASCII character G in the output, you could place its decimal equi-

valent of 71 in the Format string as \071.

V1, V2, ...

Required. Any expressions giving the values to be output in the form described by the Format parameter. Each of the Vn parameters is evaluated and written in the order in which each appears in the parameter list. The way in which they are formatted is dictated by the % format specifications. V1 is formatted by the first % sequence in the Format parameter, V2 by the second, and so on. If there are more V parameters than % sequences in the Format string, the remainder are ignored. If there are fewer V parameters than % sequences in the Format string, the remaining % sequences are written literally without any translation. If a numeric value is Invalid or outside of the range of the type indicated by the format specifier, a 0 is used as its value. If a text string value is Invalid, spaces will be output. Invalidity of Vn parameters does not preclude execution of this function.

Comments:

This function is a layered graphics statement. See Use Scaling to Position GUI Objects for information about positioning a layered graphic.

The Left and Right references are interchangeable. Whichever is smaller is taken as the left and the larger of the two values will be used as the right. The same is true of the top and bottom references. Note that the 1st 42 pixels of a VTScada application will be obscured by the title bar, if present.

It produces formatted text in exactly the same manner as BuffWrite, FWrite, and SWrite. The text is displayed directly on the screen, centered in the reference box after any animation is applied.. Binary formats aren't useful here. They will produce strings with non-printable and non-ASCII characters in them (they will be unreadable).

Example:

```
GUIText(0, 100, 100, 0 { Bounding box of text },
        1, 1, 1, 1, 1 { No scaling },
        0, 0 { No trajectory or rotation },
        1, 0 { Text is visible; reserved },
        0, 0, 0 { Cannot be focused/selected },
        Brush(12, 0, 1){ Brush is red, background
                         ignored (style is solid) },
        Pen(15, 1, 1) { White solid line 1 pixel wide },
        Font("Courier" { Font name },
              0 { Character set },
              14 { Height },
              0 { No rotation },
              0 { Weight },
              0 { Not italicized },
              1 { Fixed pitch }),
        4 { Center horizontally },
        4 { Center vertically },
        "Valve = %6.2f%% open." { Format specifier },
        valvePosition { First value });
```

This shows text in the upper left corner of the window. It cannot be focused. No scaling or animation is performed. If valvePosition contains the number 56.789, the text is displayed as:

```
Valve = 56.78% open.
```

The first four parameters must be constants. See GUITransform for an example of how to compute the position dynamically.

Related Functions:

BuffWrite | Font | FWrite | NextFocusID | Normalize | Point | SWrite | TextAttribs | VStatus | Trajectory | ZText

Related Information:

See: "Best Practices for Graphics" in the VTScada Programmer's Guide

GUITransform

Description:	Applies a graphical transformation to all graphics in a module and returns an indication when selected by a mouse button or the <ENTER> key.
Returns:	GUI Object Return Codes
Usage:	Steady State only.

Function Groups: Graphics, Advanced Module

Format: GUITransform(LeftReference, BottomReference, RightReference, TopReference, ScaleLeft, ScaleBottom, ScaleRight, ScaleTop, ScaleWhole, Trajectory, Rotation, Visibility, Options, Button, FocusID, FocusTrigger, ModuleCall)

Parameters:

LeftReference

A constant number that gives the left side reference coordinate. It must be a constant. A variable or expression is not valid here.

BottomReference

A constant number that gives the bottom side reference coordinate. It must be a constant. A variable or expression is not valid here. The top and bottom references are measured down from the top of the screen.

RightReference

A constant number that gives the right side reference coordinate. It must be a constant. A variable or expression is not valid here.

TopReference

A constant number that gives the top side reference coordinate. It must be a constant. A variable or expression is not valid here.

ScaleLeft

Required. Either a numeric expression, or any expression that returns a Normalize value. This parameter scales this side from its reference position with respect to the opposite side. If it is a numeric expression, a value of 1 will place the side at its reference position. A value of 0 will place it at the opposite side reference position. Similarly, a Normalize value will scale the

side between the high and low limits. If the value is at the high level, the side will be at its reference position. If the value is at the low level, the side will be at the opposite side reference position.

ScaleBottom

Required. Either a numeric expression, or any expression that returns a Normalize value. This parameter scales this side from its reference position with respect to the opposite side. If it is a numeric expression, a value of 1 will place the side at its reference position. A value of 0 will place it at the opposite side reference position. Similarly, a Normalize value will scale the side between the high and low limits. If the value is at the high level, the side will be at its reference position. If the value is at the low level, the side will be at the opposite side reference position.

ScaleRight

Required. Either a numeric expression, or any expression that returns a Normalize value. This parameter scales this side from its reference position with respect to the opposite side. If it is a numeric expression, a value of 1 will place the side at its reference position. A value of 0 will place it at the opposite side reference position. Similarly, a Normalize value will scale the side between the high and low limits. If the value is at the high level, the side will be at its reference position. If the value is at the low level, the side will be at the opposite side reference position.

ScaleTop

Required. Either a numeric expression, or any expression that returns a Normalize value. This parameter scales this side from its reference position with respect to the opposite side. If it is a numeric expression, a value of 1 will place the side at its reference position. A

value of 0 will place it at the opposite side reference position. Similarly, a Normalize value will scale the side between the high and low limits. If the value is at the high level, the side will be at its reference position. If the value is at the low level, the side will be at the opposite side reference position.

ScaleWhole

Required. Either a numeric expression, or any expression that returns a Normalize value. This parameter scales the horizontal and vertical dimensions by the specified factor before the left, bottom, right and top coordinates are scaled.

Trajectory

Required. Either a Trajectory function, a variable containing a Trajectory value, or a numeric expression. If this is a Trajectory value or function, the appropriate translation is applied to the image after the rotation is applied. If it is a valid numeric expression, the image isn't translated, but is displayed. Any other value is Invalid.

Rotation

Required. Either a Rotate function, a variable containing a Rotate value, or a numeric expression. If this is a Rotate value or function, the appropriate rotation is applied to the image before the trajectory is applied. If it is a valid numeric expression, the image is rotated clockwise the number of degrees specified. Any other value is Invalid.

Visibility

Required. Any logical expression. If true, the image is drawn normally. If false, the image is not drawn.

Options

Bit 1 (2^1) == Marks the transform as a

GUIStretch. Rendering of the transform is done by stretching the content.
Bit 2 (2^2) == Transform may be dragged onto a window from the palette.
All transforms in a palette that can be dragged must have both bits set. A transform may be dragged even if its visibility flag is set to zero.
Note that only Windows with bit 22 of the Style parameter set can function as a drop-target.
Windows with bit 23 of the Style parameter set are palette windows.

Button

Required. Any numeric expression giving the button combination that activates this graphic.

Value	Locator Buttons
0	No buttons
1	Right button
2	Middle button
3	Right and middle buttons
4	Left button
5	Left and right buttons
6	Left and middle buttons
7	All three buttons

If the above values are multiplied by 8, the meaning for multiple buttons pressed becomes "OR" rather than "AND." For example, to accept any button on a 2 or 3 button mouse, use 56 (i.e. 8 * 7). To accept the left mouse button regardless of whether or not the right button is pressed, use 32 (i.e. 8 * 4).

If a 64 is added to this parameter, the function will become true when the mouse buttons are released rather than when they are pressed.

FocusID

Required. Any numeric expression giving the focus number of this graphic. If FocusID is 0, this graphic cannot receive the input focus. This parameter's value may be used in a NextFocusID statement to force this graphic to get the focus.

FocusTrigger

Required. Any logical expression. If FocusTrigger changes from a valid false to a valid true, this graphic will attempt to obtain focus.

ModuleCall

Required. Any function containing a module call.

Comments:

This function is a layered graphics statement. See "Use Scaling to Position GUI Objects" for information about positioning a layered graphic.

The Left and Right references are interchangeable. Whichever is smaller is taken as the left and the larger of the two values will be used as the right. The same is true of the top and bottom references. Note that the 1st 42 pixels of a VTScada application will be obscured by the title bar, if present.

This is typically used to display a module which contains an active graphics symbol.

The transformation is applied by finding the minimum area box which contains all reference box coordinates (the first 4) for all graphics active in that module. A scaling is performed on all graphics in that module to make them appear in the reference box specified in the transform.

This is typically used to make a group of objects move and scale together. For example, a module (called PumpSymbol) could draw a symbol (such as a pump). By applying a transform to this module, it can be made to appear in a certain area of the screen.

Any mouse functions contained by the transform will also be affected. For example, if the object is rotated 90 degrees XLoc will begin to behave like YLoc.

A draggable GUITransform has two renderings. The palette rendering is what is displayed in the palette and is the image attached to the mouse at the start of dragging the transform onto a drop-target window. When the mouse cursor is moved over a drop-target window, the module called by the GUITransform is rendered at the native size of the called module and displayed in its place. The rendering is scaled according to the zoom factor of the drop-target window. Moving off the drop target window again displays the palette rendering.

The relative mouse position within the graphic when the drag operation is started is maintained throughout the operation. Therefore, when the image switches from the palette rendering to the called module rendering, the mouse position within the called module rendering is over the same feature of the graphic as when the palette rendering is used.

Examples:

```
GUITransform(0, 150, 100, 50 { Bounding box of object },
             1, 1, 1, 1, 1 { No scaling },
             0, 0 { No trajectory or rotation },
             1, 0 { Object is visible; reserved },
```

```
            0, 0, 0 { Graphic cannot be focused },
            flow = PumpSymbol(1, amps) { A sample module call });
```

This shows a pump object in the upper left corner of the window. It cannot be focused. No extra scaling is performed, and no animation is performed.

This shows how a symbol can be defined once, and positioned in a multiple area (reused).

To use the magic formula to situate a button at a position given by left, bottom, right, top:

```
If GUITransform(0, 1, 1, 0 { Unit bounding box },
               1 - (left) { Left scaling },
               bottom { Bottom scaling,
               right { Right scaling },
               1 - (top) { Top scaling },
               1, 0, 0 { No overall scaling/movement },
               1, 0 { Object is visible; reserved },
               0, 0, 0 { Graphic cannot be focused },
               flow = PumpSymbol(1, amps) { A sample module call });
[
...
]
```

The following displays a GridList that will re-size to match the Window object it is displayed within:

```
BOT = VStatus(Self(), 12);
TOP = VStatus(Self(), 11);
GUITransform(0, 1, 1, 0   { Reference rectangle         },
             1- 0 { Left edge of window },
             BOT { Bottom edge of window },
             TOP { Right edge of window },
             1- 0 { Top edge of window },
             1                  { No overall scaling          },
             0, 0, 1, 0         { No movement; visible; res   },
             0, 0, 0            { Not selectable              },
             \System\GridList(GridTitles  { Titles array     },
                              AllList { Data array          },
                              Formats { Cell format array   },
                              ColWidths  { Column widths array   },
                              PickValid(ArraySize(AllList,0), 0) {
rows },
                              3      { # Data cols          },
                              15, 8  { Grid BGnd, Grid color },
                              1, 1 { Grid line width, line style},
                              30, 30 { Row/Title height      },
                              0, 0 { Horiz/Vert cell padding },
                              0, 0 { Horiz/Vert Scroll position },
                              0, 0 { Enable V/H scroll bars  },
                              0    { Enable column sizing    },
                              1    { Disable Sorting         },
                              1    { Disable selected cell   },
```

```
                                0, 0 { Disable V/H grid lines  },
                                1    { LockFirstColumn         },
                                Invalid  { Sort                },
                                0        { SelectedRow         },
                                0        { SelectedColumn      },
                                \_DialogFont { GridFontParm    }));
```

Related Functions:

GetXformRefBox | NextFocusID | Normalize | Point | Rotate | Trajectory | UnTransform | Vertex | VStatus | Window

Related Information:

See: "Best Practices for Graphics" in the VTScada Programmer's Guide

H Functions

The sections that follow identify all VTScada functions beginning with "H".

HasCompilationErrors

Description: Reports if the working copy presently has unresolved compilation errors

Returns: Boolean

Usage: Script Only.

Function Groups: Configuration Management

Related to: IsAppEditable |

Format: LayerModule\HasCompilationErrors()

Parameters: None.

Comments: Returns TRUE if the last attempted compilation of the layer failed. This is a situation that shouldn't ever be encountered, as any attempt to commit a compilation error to the repository should fail, with the working copy (WC) being rolled back to a compilable version.
However, if the WC becomes corrupted or if there are com-

pilation dependencies outside of the working copy, then in theory the WC can be in an unresolvable, non-compilable state, which is what this subroutine reports. This function should only be called as a subroutine, following Activation.

Hash

Description: Generates a hash – a text string of bytes – of the given string.

Returns: Text

Usage: Script only.
May be used in optimized Tag Parameter Expressions.

Function Groups: Cryptography

Related to: Encrypt | Decrypt

Format: Hash(PlainText[, AlgID, Seed, HashHandle])

Parameters:

PlainText

Required. Any text string to create a hash from.

AlgID

Specifies the algorithm to use when creating the hash. Valid options are as follows: Defaults to 0 (SHA-1) if missing or invalid.

AlgID	Hash
0	SHA-1 (160 bit hash)
1	MD5 (128 bit hash)
2	SHA-2 (256 bit hash)
3	SHA-2 (384 bit hash)
4	SHA-2 (512 bit hash)

Seed

Optional text buffer, which will be used as a seed for

the hash algorithm. The contents of the buffer will be pre-pended to the text from the first parameter when creating the hash.

HashHandle

Optional. LValue. If Invalid, a hashing handle for the given AlgID is created and used for this and subsequent calls that supply this handle. No output is generated until a call, with this handle, is made with an Invalid PlainText parameter, at which time the hash built up using the handle is returned. At that time, the handle is destroyed and the LValue invalidated.

For one-time hashing, the hash value is returned from this function as a text string of bytes. For progressive hashing the return value of this function is Invalid until a call with an Invalid PlainText parameter and a valid HashHandle is made.

Comments The use of a seed string is functionally equivalent to the following example, but will execute slightly faster.
X = Hash(Concat("XYZ", "ABCDE"), 2);.

HasMetaData

Description: Tests whether a given variable is a dictionary. Since the default behavior of most operands and functions on dictionaries is to return just the value of the dictionary's root, this function provides the only means to determine whether or not a variable contains a dictionary.

Returns: Boolean

Usage: Script or steady state.

Function Groups: Dictionary, Variable

Related to: Dictionary | MetaData | IsDictionary

Format: RVAL = HasMetaData(dictionary);

Parameters:

Dictionary

Required. The name of any variable to test.

Comments: Returns TRUE if the object is a dictionary and FALSE otherwise.
Since the default behavior of most operands and functions on dictionaries is to return just the value of the dictionary's root, this function provides the only means to determine whether or not a variable contains a dictionary.

HasReturnStatement

Description: Examines a specified object to see if it is running a return statement in steady-state.

Returns: Boolean

Usage: Script Only.

Function Groups: Basic Module

Related to: SetReturnValue

Format: HasReturnStatement(Object)

Parameters:

Object

Required. Reference to the object of interest.

Comments: This function examines a specified object to see if it is running a return statement in steady-state. Returns 1 if such a statement is running, 0 if no return statement is running in steady-state, or Invalid if the first parameter cannot be resolved to an object.

HasUndeployedChanges

Description: Finds whether the local machine is maintaining changes that have not been deployed, including changes that have

been recorded by EditFile but have yet to be committed.

Returns: Boolean

Usage: Script or steady state.

Function Groups: Configuration Management

Related to: HasCompilationErrors | IsAppEditable | IsRunOnly |

Format: \LayerRoot\HasUndeployedChanges

Parameters: none

Comments: This function returns TRUE if the Layer is running with changes that have not been deployed. Either it is on a local branch, or there are working copy modifications that have yet to be committed such as page changes.
Note that the return value doesn't get updated while we are within the } { working copy lock so that atomic (within the working copy lock) commits do } { not cause the return value to flicker.

Help

Description Calls a topic in a help system.

Returns Nothing

Usage Script Only.

Function Groups Help

Related to: SetHelp

Format Help(FileName, Option, SearchValue)

Parameters

FileName

Required. Any text expression for the name of the help system to be displayed. May be the name of a .CHM file, or a.HLP file. To call a topic in the VTScada help system, use the system prop-

erty, \DevHelpFile.

Refer to the comments section for more options.

Option

Required. Any numeric expression controlling how the topic will be located within the help system, as indicated by the following table. Not all options work with all help system formats. Calls to topics using context sensitive help ID values should use option 3.

Option	Display Option	SearchValue
0	Help contents	0
1	Key	Key string
2	Partial key	Partial key string
3	ID	ID number
4	Help Topics	0

SearchValue

Required. Any expression that provides the proper qualification needed by the Option parameter as indicated by the preceding table.

Comments

If calling a help system compiled using the DocToHelp NetHelp system, use the string, "MyHelpfolder\NetHelp". If calling your own help system compiled using the Flare HTML5 system, use the string "MyHelpFolder\MadCapWebHelp".
DocToHelp and Flare are products of Madcap Software Ltd. and are recognized by VTScada.

Example:

```
If ZButton(10, 40, 110, 10, "Help", 1);
[
  Help("C:\VTScada\MyCustomHelpFile.chm", 0, 0);
]
```

This displays a button on the screen that when pressed, opens the table of contents of the named CHM format file.

```
If WinButton(513, 135, 547, 101,
             0 { normal appearance },
             "?" { label },
             1 { focus id enables button },
             0 { default system font },
             Invalid { no ToggleVal used },
             Invalid { image displayed in the button});
             [
                Help(\DevHelpFile,3,110);
             ]
```

This will display a button that when pressed will open topic id 110 in the VTScada help system. Note that 110 is one of the ID values set aside for user-created topics.

Related Information:

See: "User-Topics in the VTScada Help Folder" in the VTScada Developer's Guide.

HexToBuff

Description: Converts a hex string to a binary buffer.

Returns: Buffer

Usage: Script only.
May be used in optimized Tag Parameter Expressions.

Function Groups: String and Buffer

Related to: BuffToHex

Format: HexToBuff(HexValue)

Parameters:

HexValue

Required. Any string representing a hexadecimal value.

Comments: Given an invalid parameter value, the function will return Invalid.

Examples:

```
BuffResult = HexToBuff("57494C4C");
```

BuffResult will contain "Will".

HighlightModule

Obsolete. Code using this function will compile, but will do nothing.

Description Highlighted a module in a module tree.

Returns Boolean

Usage Script Only.

Function Groups Module Tree Diagram

Related to: HighlightState | HighlightTree | ModuleHighlighted

Format HighlightModule(ModuleTree, Module, Mode)

Parameters

ModuleTree

Required. Any expression for the module tree value.

Module

Required. Any expression for the code value of the module.

Mode

Required. Any numeric expression that indicates how to highlight

Mode	Highlight
0	Turn on highlighting
1	Turn off highlighting
2	Toggle highlighting

Comments none

HistorianConnect

Description Opens a logging connection and controls the lifetime of all resources associated with that connection.

Warning This function should be used only by advanced programmers. The functions, WriteHistory and GetTagHistory and GetLog are recommended for most uses.

Returns Invalid if the parameters are invalid, 0 otherwise

Usage Script Only.

Threaded Yes

Function Groups Logging

Related to: HistorianDeleteRecords | HistorianGetData | HistorianGetInfo | HistorianReadRecords | HistorianWriteRecords

Format HistorianConnect(ConnectionType, ConnectionString, HistorianName, Namespace, HistorianHandle, Error)

Parameters

ConnectionType

A text keyword that indicates the storage methodology required. Permitted values are

- FileDB for storage on a local or networked hard drive, using the VTScada proprietary database format.
- ODBC for storage using a local or networked DBMS, accessed via ODBC drivers.

The keywords are not case-sensitive.

ConnectionString

A text string that provides the information required to connect to the database specified by the ConnectionType.

ConnectionType	ConnectionString
FileDB	The root path to the storage folder. If a relative path is provided, then it is relative to the VTScada installation directory
ODBC	Any valid ODBC connection string. This may begin with DSN= to specify a data source name, or it may be a complete connection string.

HistorianName

A text string providing the Historian Tag Name. This string is used as part of the full path to a file-based data store and is also the name of the UpTime table in the database.

Namespace

The top-level namespace in which to store data (may be an empty string).

HistorianHandle

Historian Connection handle value. For the VTScada proprietary data store, this will be invalidated on an "out of disk space" error, or on loss of access to the file storage. For other databases, this will be invalidated on any connection loss.

Error

A value in which the error code, (if any) will be returned.

Comments The connection to the data store is done asynchronously. HistorianHandle will become valid when the connection has been made. If the connection fails Error will be set to some sort of error information.
Each separate connection will create its own thread for writing data. Invalidating the connection handle will flush all pending write data and then terminate the thread, DB connections etc.
The Namespace parameter may be used to allow the same root storage location to be shared by multiple applications or machines or both.

HistorianDeleteRecords

Description: Conditionally deletes full sequences of records from the data store identified by the ConnectionHandle

Warning: This function should be used only by advanced programmers. The functions, WriteHistory and GetTagHistory and GetLog are recommended for most uses.

Returns: An array of SequenceID values, indicating which sequences were deleted

Usage: Script Only.

Threaded: Yes

Related to: HistorianDeleteRecords | HistorianDeleteRecords HistorianGetData | HistorianGetInfo | HistorianReadRecords | HistorianWriteRecords

Format: HistorianDeleteRecords (ConnectionHandle, TagName, Schema, MinGroupValue, NumRecordsToKeep)

Parameters:

ConnectionHandle

As returned by a HistorianConnect call.

TagName

A text string identifying the tag whose records are to be deleted.

Schema

This is a textual representation of a schema structure.

MinGroupValue

All sequences with a Group Value (the last 4 bytes of the 12-byte SequenceID) less than this value will be deleted, provided that the NumRecordsToKeep condition is also met.

NumRecordsToKeep

A numeric expression, specifying a minimum number of records that are to be kept for the tag.

Comments: This function will delete as many entire sequences from storage for a tag as it can so that at least NumRecordsToKeep records are kept, and only sequences with a lower group value than MinGroupValue are deleted. While considered to be threaded, this function has no lifetime once it passes the records to another thread for deletion.

HistorianGetData

Description: Queries a Historian data store and returns an array of processed records.

Returns: Invalid if parameters are invalid, otherwise 0.

Usage: Script Only.

Threaded: Yes

Related to: HistorianGetData | HistorianDeleteRecords | HistorianGetData HistorianGetInfo | HistorianReadRecords | HistorianWriteRecords

Format: HistorianGetData (ConnectionHandle, TagName, Schema,

Data, FieldNames, TPP, StartTime, EndTime, NumEntries, Modes, StaleTimes, LastRestartTime, ErrorCode)

Parameters:

ConnectionHandle

As returned by a HistorianConnect call.

TagName

A text string representing the desired table name. Engine code may modify this to conform with storage limitations.

Schema

A text expression describing the schema from which to retrieve data.

Data

A variable in which the data will be returned.
This will be a two-dimensional array representing the returned data, indexed by [col][row]. If TPP is non-zero, the number of rows in the result set will equal the value of the NumEntries parameter. If TPP is zero there could be fewer rows than specified by NumEntries

FieldNames

Names of the fields (as opposed to the number in TGET). There is no need to support time options as in TGET.

TPP

Required. Any numeric expression giving the time span in seconds for each array entry. Each array element will contain the data which correspond exactly to this time period which corresponds to 0 or more data points. If TPP is positive and FieldNames selects a text value, the first entry which falls in a time is read and Mode is ignored.
If TPP is equal to 0, the data is read and placed in the

array on a one to one basis.
If TPP is less than 0, an error will be returned.

StartTime

Timestamp in UTC. The report will include values matching and later than this time.

EndTime

Ending timestamp. Ignored if TPP is non-zero. The report will include values less than and matching this value.

NumEntries

Number of log entries in the array (numEntriesRequested - determined by Size or TPP parameters of Getlog)
Use a negative value to retrieve values in reverse chronological order.

Modes

Optional numeric expression giving the method of handling the data. If TPP is greater than 0, the values that fall in each time span will be represented as follows:

Mode	Time Span Representation
0	Time weighted average
1	Minimum in range
2	Maximum in range
3	Change in value over the range
4	Value at start of range
5	Time of minimum in range (in seconds since Jan 1, 1970)
6	Time of maximum in range (in seconds since Jan 1, 1970)
7	Count the total number of zero to non-zero transitions within each TPP period.
8	Totalizes, for each TPP, the amount of time when the value is non-zero (Invalid is counted as zero).
9	Totalizes, for each TPP, the arithmetic sum of the recorded values.
10	Interpolates between values.

In the case of modes 5 and 6, FieldName should still be set to indicate the field on which the mode is to act; the return values will be times indicating the maximum or minimum in that field for each time span.

If TPP is less than or equal to 0, Mode is ignored. If the data is text, the first entry in a given time range is used for the array entry and

Mode is ignored.
It is possible to retrieve more than one mode in a single GetTagHistory statement. To do this, pass an array of values in as the Mode parameter.

StaleTimes

An optional parameter that sets a maximum validity duration for data elements that are being TPP processed. Normally, every data point is treated as remaining valid until the next data point. If a valid StaleTime parameter is given, then any data point will be treated as invalid StaleTime seconds after the recorded time. If TPP is less than or equal to 0, StaleTime is ignored. If StaleTime is not required but EnableDowntimeOverride is, then StaleTime should be given as an Invalid value. It is possible to specify more than one stale time in a single GetTagHistory statement. To do this, pass an array of values in as the StaleTime parameter.

LastRestartTime

A Timestamp value used in UpTime processing. (see comments)

ErrorCode

A value in which the error code (if any) will be returned.

Comments: If a record has been written that duplicates the timestamp of an existing record then, when reading, the newer record will be supplied in place of the older. This arbitration is performed by means of a Secondary (or Generation) Timestamp.

To support this, properties can be associated with Schema columns. One property defines the Primary

Timestamp for the record and another property value defines the Secondary (or Generation) Timestamp for the record. If no secondary timestamp column is defined, then arbitration will not take place. If no primary timestamp column is defined, then retrieval of a specific time range is not possible.

Uptime processing allows VTScada to show gaps in data for periods when VTScada did not record any data. As long as there is at least one Historian logging on a machine, the logging system is deemed available and there is no downtime. Any gap indicates that the logging system was down. (That is, the application may be running but the Historian has lost its connection to its SQL database). The UpTime information is logged by a Historian tag and is therefore to be found in the Historian's normal log data table.
No downtime invalids are inserted when reading in the reverse order.

HistorianGetInfo

Description: Retrieves information about a specified Historian or all Historians.

Returns: Varies depending on the request.

Usage: Script Only.

Threaded: No.

Related to: HistorianGetInfo | HistorianDeleteRecords | HistorianGetData | HistorianGetInfo HistorianReadRecords | HistorianWriteRecords

Format: HistorianGetInfo(ConnectionHandle, InfoType)

Parameters:

ConnectionHandle

As returned by a HistorianConnect call.

InfoType

InfoType	Information requested
0	Counters. A dictionary of dictionaries of dictionaries containing full counter information for all sequences stored by this Historian.
1	Errors. A dictionary of write or deletion errors that occurred in this Historian since the last time GetInfo (Historian, 1) was called.
2	Overall write queue count. The number of records in the write queue for all Historians (ConnectionHandle parameter is ignored).
3	Overall records stored. A rollover count of the records stored by all Historians (ConnectionHandle parameter is ignored). This counter rolls over (2^32) and therefore is only useful for performance diagnostics. It is not a count of all records in storage.
4	Write queue count. The number of records in the write queue for this Historian.
5	Records stored. A rollover count of the records stored by this Historian. This counter rolls over and therefore is only useful for performance diagnostics. It is not a count of all records in storage.
6	Overall write queue size. The approximate memory size of the write queue for all Historians (ConnectionHandle parameter is ignored).

HistorianReadRecords

Description: Reads a range of records from the data store. The range is specified by the Start and End counter values supplied.

Warning: This function should be used only by advanced programmers. The functions, WriteHistory and GetTagHistory and GetLog are recommended for most uses.

Returns: Invalid if parameters are invalid, 0 otherwise.

Usage: Script Only.

Threaded: Yes

Related to: HistorianReadRecords | HistorianDeleteRecords | HistorianGetData | HistorianGetInfo | HistorianReadRecords HistorianWriteRecords

Format: HistorianReadRecords (ConnectionHandle, TagName, Schema, Data, MID-S, StartCounter, EndCount, ErrorCode)

Parameters:

ConnectionHandle

As returned by a HistorianConnect call.

TagName

A text string representing the desired table name. Engine code may modify this to conform with storage limitations.

Schema

This is a textual representation of a schema structure.

Data

A variable in which the values will be returned. This will be a single dimension array representing the Fields (columns). The entries in the array will be an array of values (all fields will be the same) and the size of one of these arrays gives the number of records (rows) in the result set.

Mid-S

A 64 bit binary token.

StartCounter

Starting counter number. Starts counting at 1 rather than zero.

EndCounter

Ending counter number. Starts counting at 1 rather than zero.

ErrorCode

FileDB or SQL DB error code returned.

Comments: Data will be set on completion of the operation. ErrorCode will be set on completion of the operation. This will be set to zero if the operation was successful

Together, the Schema and the MID-S parameters define the subset of records in the Index file from which the data in the required Counter range will be retrieved. If this filtered subset has no data, then none will be returned.

HistorianWriteRecords

Description Writes records to the data store identified by the ConnectionHandle

Warning This function should be used only by advanced programmers. The functions, WriteHistory and GetTagHistory and GetLog are recommended for most uses.

Returns A counter value, or Invalid if parameters are invalid

Usage Script Only.

Threaded Yes

Related to: HistorianWriteRecords | HistorianDeleteRecords | HistorianGetData | HistorianGetInfo | HistorianReadRecords

Format

HistorianWriteRecords (ConnectionHandle, TagName, Schema, Records, SequenceID)

Parameters

ConnectionHandle

As returned by a HistorianConnect call.

TagName

A text string representing the desired table name. Engine code may modify this to conform with storage limitations.

Schema

This is a textual representation of a schema structure.

Records

Array of data. This is a single dimension array representing the Fields (columns) to be written. Each entry in this array may be either a single data value or an array of values. If any entry is an array, then the size of the largest array gives the number of records (rows) to be written. If any individual field is underspecified, then a representation of INVALID is written for the missing values.

SequenceID

A 12-byte, binary token.

Comments

This is the normal method of writing historical data. Data is always appended to the data store, and records are not deleted or replaced unless HistorianDeleteRecords is called explicitly.
The function returns the counter value for the last record being written.

Note: At the point that the function returns, the data has not yet been written. Since the Counter is the actual record number (FileDB) or an index (SQL) the counter value

returned here will be a calculated value, based on the number of records that are pending.

As well as defining the column format, the Schema defines the subset of tables to which the supplied data will be written.

While considered to be threaded, this function has no lifetime once it passes the records to another thread for writing.

Each record will be represented by a Counter value which is one greater than the previous Counter value, unless either Schema or MID-S have changed from the previous write. In this case, a new table/file set is started. The counter value for a new table always starts from 1.

HScrollbar

(System Library)

Description Draws a horizontal scrollbar and returns its position.

Returns Boolean

Usage Steady State only.

Function Groups Graphics

Related to: ToolBar | VScrollbar

Format \System\HScrollBar(Left, Top, Width, Steps, PageLen[, Offset, StepSize, WindowObj])

Parameters

Left

Required. Any numeric expression for the left side coordinate of the scrollbar.

Top

Required. Any numeric expression for the top coordinate of the scrollbar.

Width

Required. Any numeric expression for the width of the scrollbar in pixels.

Steps

Any numeric expression giving the number of steps in the scrollbar.

PageLen

Required. Any numeric expression giving the number of steps to jump per page.

Offset

Optional. Any numeric expression giving the exported/imported scroll position. Defaults to 0 if not specified.

StepSize

Optional. Any numeric expression giving the number of lines to scroll through when the user clicks on an arrow. Defaults to 1 if not specified.

WindowObj

Used by the Anywhere client. Enables redirection to the HScrollbar of horizontal panning over a given region. Required because on many touchscreens you cannot interact directly with the scrollbar therefore this object is needed to make it possible to scroll HScrollbar on those platforms.

Comments

This module is a member of the System Library, and must therefore be prefaced by \System\, as shown in "Format" above. If you are developing a script application, use "System\..." rather than "\System\..." in the function call. Steps can be calculated by subtracting the number of visible items from the total number of scrollable items, while PageLen is equal to the number of visible items.

Example:

```
\System\HScrollBar(VStatus(Self, 11) - VStatus(Self, 21) { Left },
                   VStatus(Self, 12) - 1 { Top },
                   VStatus(Self, 12) { Width },
                   Length - LinesVisible { Total steps },
                   LinesVisible { Steps in page },
                   Offset { Thumb tab pos },
                   StepSize { Number of lines per arrow click. });
```

I Functions

The sections that follow identify all VTScada functions beginning with "I".

IconMarker

Note: Deprecated. Do not use in new code.

Description: Creates the question mark and exclamation mark graphics, used to indicate questionable and manual data in a widget.
This module places a set of icons on the screen centered over a given rectangular region. The icon displayed is cycled with each passage of the period, measured in seconds

Returns: Nothing

Usage: Steady State only.

Function Groups: Graphics

Related to: GetXformRefBox | Scale | TagIconMarker

Format: IconMarker(Left, Bottom, Right, Top, Period, Mode, SizeX, SizeY, IconArray)

Parameters:

Left, Bottom, Right and Top

Are numeric expressions providing the bounds of the graphic to mark. The symbol will appear in the center of this area.

Period

A numeric expression setting the period (in seconds) at which the symbol is changed

Mode

A numeric expression indicating the symbol to display according to the following table

Mode	Symbol
0	Inhibit symbol display
1	Questionable Data symbol
2	Manual Data symbol
3	Questionable and Manual symbols
4	Error symbol
5+	User supplied symbols

SizeX

A numeric expression providing the horizontal extent of the symbols to be displayed

SizeY

A numeric expression providing the vertical extent of the symbols to be displayed.

IconArray

The array of images to be displayed. This must be a variable – use of the "Invalid" keyword or a constant here will cause the module to fail. The value of this variable is discarded for mode values less than 5.

Examples:

To load an image into IconArray for use with mode 5, you might use code similar to the following. The image should be 12 x 12 pixels.

```
CIcon   = 0;
IconArray[CIcon] = PickValid(MakeBitmap("BitmapName.bmp", 1),
\IMStandardArrays[3][0]); { Color 1 is set as transparent }
```

To use IconMarker in a tag widget, you must check the value of the tag's questionable and manual data parameters. The code may be similar to the following.
(example taken from the TopBar widget. Note that in this example, the vertical position of the icon changes with size of the TopBar.)

```
  Call(Variable((PickValid(\Questionable, 0) || Valid(\ManualValue))?
("IconMarker"):(Invalid)),
                GetXFormRefBox(Self(), 0),
                GetXFormRefBox(Self(), 1),
                GetXFormRefBox(Self(), 2),
                Scale(SVal, SMin, SMax, GetXFormRefBox(Self(), 1),
GetXFormRefBox(Self(), 3)),
                1,
                ((PickValid(\Questionable, 0))?(1):(0)) + ((Valid
(\ManualValue))?(2):(0)),
                12, 12, MarkList);
```

(MarkList is a variable declared in the widget as a holder for the markings.)

IF

Description: Performs an action; it changes the active state in a module instance, or executes a script, or both.

Returns: Nothing

Usage: Steady State only.

Function Groups: Logic Control

Related to: IfElse | IfOne | IfThen | Cond

Format:

```
IF Trigger Destination;
[
  Script
]
```

Parameters:

Trigger

Required. Any logical expression. If false, nothing happens. If true, the action is performed, and any functions that may be reset are automatically reset.

Destination

Optional. Destination is not an expression, it is the text name of a state. If provided, it must be a legal name of a state in the module where this action is defined.
It is not possible to change to a state in another module; each module must have one (and only one) active state, unless the module has no states defined.

Comments: This is the only way to change the active state in a module. An action may have a Destination, a Script, or both, or neither (although it does not make sense to have neither).

Script is a list of functions and statements, ending in semicolons, contained by square brackets. The Script is optional; it may be omitted. There is no limit to the number of statements in Script (other than RAM). If the script is omitted, the square brackets must also be omitted. A script's statements are executed in order, from top to bottom. While a script is executing, no other statements, functions, or scripts may execute until this script is complete.

Examples:

```
highLevel = 0;
...
FillUp
[ { Begin state FillUp }
  If level > 36 heaterOn { Wait for tank to fill };
] { End state FillUp }
```

This action is triggered if the variable level is a number greater than 36. If this action is triggered, the state FillUp is stopped, and the state HeaterOn is started.

```
{ An example of poorly written code }
HeaterOn
[ { Begin state HeaterOn }
  If level > 40 { Monitor the level };
  [ { Begin script }
    highLevel++;
    highAlm = highLevel > 10 { Sound high level alarm };
  ] { End script }
] { End state heaterOn }
```

This is an example of what to AVOID!

If level rises above 40, this action will trigger over and over again. This is called an "If 1" condition and will degrade system performance easily to 10% or 1% or less of its normal potential. All the time level > 40 is true, highLevel will increase by one at a very rapid rate. This is because there is no destination state; the action remains active, and the trigger is checked again and again.
Note also that the script has two statements. They are executed once only, in order; this means that the script will have to execute 10 times before the first statement has increased highLevel enough so that the second statement will execute.
If what is desired is to count the number of times level rises above 40, a better way to write this would be:

```
HeaterOn
[ { Begin state HeaterOn }
  If Change(level > 40, 0) && level > 40
  { Monitor the level };
  [ { Begin script }
    highLevel++;
    highAlm = highLevel > 10 { Sound high level alarm };
  ] { End script }
] { End of state heaterOn }
```

This waits for level to change while also being above 40. Once this happens, the action is triggered, the script is executed, and the Change function is automatically reset to wait for another change. The Change function will not re-trigger until level drops below 40, when there will finally be a change in the condition level > 40. At that time, however, since level will be less than 40, the action won't trigger until level rises above 40. This is how to trigger on the rising edge of a logical expression (level > 40).
It is recommended that you refer to "Automatically Reset Functions" for further information.

IfElse

Description: Returns the result of one of two expressions depending upon the result of a conditional expression.

Returns: Varies

Usage: Script or steady state.

Function Groups: Logic Control

Related to: | Cond | Execute | IfThen

Format: IfElse(Condition, TRUECase, FALSECase)

Parameters:

Condition

Required. An expression that returns true or false. If true the TRUECase is executed. If false, the FALSECase is executed. If invalid, neither case is executed.

TRUECase

Required. The expression (typically an Execute statement) which is executed when Condition is true.

FALSECase

Required. The expression (typically an Execute statement) which is executed when the Condition is false.

Comments: This is simply a different name for the Cond function. The return value is the result of the evaluated case. Both cases are evaluated, in which case Cond is a more apt name.

Example:

```
If 1 Main;
[
  IfElse(i > 0 && i <= 50,
        Execute(y[i] = x,
        i++),
  { else }
        Execute(x = i,
                doneFlag = 1)
  );
]
```

This statement will set the element of array y equal to i and then increment i if i is between 1 and 50 inclusive; if i is not in that range, x will be

set to i and doneFlag will go true. Note that IfElse and Execute may only appear in a script.

IfOne

Description:	Check for an If One Condition. This function checks for a race condition in an action script and returns the value of the location.
Returns:	Module
Usage:	Script only. May be used in optimized Tag Parameter Expressions.
Function Groups:	Logic Control
Related to:	| Cond | Execute | IfElse
Format:	IfOne()
Parameters:	None
Comments:	This function is used for debugging VTScada applications and returns the module, state or statement value of the last script that did not switch states, and had a true action trigger after executing its script.

Example:

```
[
  BadMod = "";
  BadModFile = "";
  BadState = "";
  BadStatement = "";
  LastIfOne;
]

Main [
  If ZButton(10, 30, 90, 10, "Go", 1, System\DefFont);
  [
    LastIfOne = IfOne();
    BadMod = Cast(Cast(LastIfOne, 11), 4);
    BadModFile = GetModuleText(LastIfOne, 0);
    BadState = StateName(LastIfOne);
    BadStatement = Cast(LastIfOne, 1);
  ]

  ZText(10, 50, "File:", 0, System\DefFont);
  ZText(10, 70, "Module:", 0, System\DefFont);
```

```
  ZText(10, 90, "State:", 0, System\DefFont);
  ZText(10, 110, "Statement:", 0, System\DefFont);
  ZText(70, 50, PickValid(BadModFile, "N/A"), 0, System\DefFont);
  ZText(70, 70, PickValid(BadMod, "N/A"), 0, System\DefFont);
  ZText(70, 90, PickValid(BadState, "N/A"), 0, System\DefFont);
  ZText(70, 110, PickValid(BadStatement, "N/A"), 0, System\DefFont);
]
```

This script will find a possible problem area (a script that may have a race condition) and set the variables badMod, badState or badStmnt to indicate its location in the application. There would probably be several statements following this script that printed out the aforementioned variables; the ones that were not valid would simply hold the text "N/A".

IfThen

Description: Conditionally Execute Statements. This statement executes a statement if the condition parameter is true.

Returns: Nothing

Usage: Script only.
May be used in optimized Tag Parameter Expressions.

Function Groups: Logic Control

Related to: | Cond | Execute | IfElse

Format: IfThen(Condition, Expression1, Expression2, ...)

Parameters:

Condition

Required. An expression that will indicate whether or not the Expression parameters are executed. If this is true, the Expression parameters will be executed.

Expression1, Expression2, ... default }

Required. Any statements to be executed when Condition is true. Any number of Expression parameters may be listed. They are executed once in the order that they are listed.

Comments: This statement may only appear in a script.

Example:

```
If 1 Main;
[
  IfThen(Valid(tankPtr) && fluidLevel > 1000,
  startPump = 1,
   activeTankNum++,
  msg = "Changing tanks...");
]
```

The statement in the script will test to see if tankPtr exists (is valid) and if fluidLevel > 1000, if these conditions are both true, the series of actions will be taken. VTScada does not use short-circuit evaluation. Both parts of the condition will always be checked (and evaluated if required).

ImageArray

Description: Reads an existing image handle and returns another one containing an image created from that handle by tiling the image a given number of times.

Returns: Image handle

Usage: Script or steady state.

Function Groups: Graphics

Related to: Crop | GUIBitmap | GUIButton | ImageSweep | MakeBitmap | ModifyBitmap

Format: ImageArray(Handle, HorizontalSpacing, VerticalSpacing, RowCount, ColCount[, Orientation, CropLeft, CropTop, CropWidth, CropHeight, Tessellate])

Parameters:

Handle

Required. The image handle to copy and modify.

HorizontalSpacing

Required. Any numeric expression for the horizontal space between each tile.

VerticalSpacing

Required. Any numeric expression for the vertical

space between each tile.

RowCount

Required. Any numeric expression for the number of tiles running vertically.

ColCount

Required. Any numeric expression for the number of tiles running horizontally.

Orientation

Optional numeric value between 0 and 15, specifying the orientation of the tiles. Defaults to zero.

Orientation Value	Meaning	Orientation Value	Meaning
0	No rotation, no flipping	8	No rotation, flip vertically
1	90-degree CW rotation without flipping	9	90-degree CW rotation and flip vertically
2	180-degree CW rotation without flipping	10	180-degree CW rotation and flip vertically
3	270-degree CW rotation without flipping	11	270-degree CW rotation and flip vertically
4	No rotation, flip horizontally.	12	No rotation, flip horizontally, then vertically

5	90-degree CW rotation and flip horizontally	13	90-degree CW rotation, flip horizontally, then vertically
6	180-degree CW rotation and flip horizontally	14	180-degree CW rotation, flip horizontally, then vertically
7	270-degree CW rotation and flip horizontally	15	270-degree CW rotation, flip horizontally, then vertically

CropLeft

Optional. Left coordinate of the cropping region. This and the following three parameters must all be specified if any are specified. Together, they apply a clipping rectangle that will be applied to the tiled image *after* tiling occurs.

CropTop

Optional. Top coordinate of the cropping region.

CropWidth

Optional. Width of the cropping region.

CropHeight

Optional. Height of the cropping region.

Tessellate

Optional Boolean. Defaults to FALSE (0). Optim-

izes drawing for a simple grid of images, but causes ImageArray to ignore spacing and transformation of the images. Does not improve the appearance of the result, but uses significantly less memory and may be marginally faster.
The result may be clipped and stretched.

Comments: More commonly used in script than in steady-state.
The cropping parameters work in contrast to the Clip function, which applies a clipping region to the image before tiling occurs.
A regular image array will allow for gaps between the tiles and some transformations of the original image. The algorithm for this is less than optimal if one wants an unspaced grid of images, such as in a wallpaper bitmap. For the latter case, set the Tessellate parameter to true.

ImageSweep

Description: Reads an existing image handle and returns another one containing an image created from that handle by tiling the image a given number of times along an arc-shaped path.

Returns: Invalid on failure

Usage: Script or steady state.

Function Groups: Graphics

Related to: Crop | GUIBitmap | GUIButton | ImageArray | MakeBitmap | ModifyBitmap

Format: ImageSweep(Handle, Count, Radius, StartAngle, EndAngle[, CropLeft, CropTop, CropWidth, CropHeight])

Parameters:

Handle

Required. The image handle to copy and modify.

Count

Required. Any numeric expression for the number of tiles to draw.

Radius

Required. Any numeric expression for the natural radius of the short axis of the ellipse, used to determine scaling.

StartAngle

Required. Any numeric expression for the angle at which the first tile is to be drawn. The range -180 to 180. The start point is the vertical axis. Angles are positive, counter-clockwise..

EndAngle

Required. Any numeric expression for the angle at which the last tile is to be drawn. Range -180 to 180, start point is the vertical axis, angles are positive counter-clockwise..

CropLeft

Optional. Left coordinate of the cropping region. This and the following three parameters must all be specified if any are specified. Together, they apply a clipping rectangle that will be applied to the tiled image *after* tiling occurs.

CropTop

Optional. Top coordinate of the cropping region.

CropWidth

Optional. Width of the cropping region.

CropHeight

Optional. Height of the cropping region.

Comments: The result is a radial array of tiles. Drawing angles

intersect the center of the tiles. The path is based upon the size and shape of an ellipse traced out using the bounding box of the drawing transform. Scaling is based upon a comparison of the described radius and the final size of the transform on the screen. In the case of an ellipse the radius is assumed to be the smallest elliptical radius. Returns Invalid upon failure
The cropping parameters work in contrast to the Clip function, which applies a clipping region to the image before tiling occurs.

ImportAPI

Description: Imports objects of class API from a given module, for use in the calling module.

Returns: Numeric

Usage: Script Only.

Function Groups: Advanced Module

Related to:

Format: ImportAPI(SourceModule[, DestinationModule])

Parameters:

SourceModule

Required. Any expression for the module from which the API-class constants should be imported.

DestinationModule

An optional expression, giving the module that the API-class constants should be imported into. Used only if the constants are not being imported into the current module.

Comments: When using modules such as TreeControl, the ImportAPI

function can reduce the programming workload by importing all the required constants with one command. Only those constants that have been declared as class API in the source module will be imported.

ImportAPI "imports" the source objects by creating variables with the same name in the destination module and assigning to these the default values from the source module. Results may be unpredictable if more than one instance of the source module is found.

Returns the number of variables that were found to have the same name in the calling module and that therefore were not imported from the source module.

Example:

A selection of constants declared in TreeControl.SRC:

```
  {***** Indices into the Tree array nodes *****}
[ (API)
  Constant #TI_KEY      = 0  { "Key" value...see heading comment };
  Constant #TI_TEXT     = 1  { Text value to be displayed        };
  Constant #TI_SUBTREE  = 2  { Subordinate tree below this node  };
… etc …
```

Importing the TreeControl constants in DropTree.SRC:

```
Init [
  If 1 DropTree;
  [
     { Import the Tree Control API }
    ImportAPI(\TreeControl);
… etc …
```

ImportKey

Description: The ImportKey function transfers a cryptographic key from a key BLOB into a CSP (cryptography service provider). It is the VTScada analog of the CryptoAPI's ImportKey call.

Returns: Key

Usage: Script Only.

Function Groups: Cryptography

Related to: DeriveKey | Decrypt | Encrypt | ExportKey | GenerateKey | GetCryptoProvider | GetKeyParam | SetKeyParam

Format: ImportKey(CSP, BlobType, KeyBLOB [, DecryptKey, Flags, Error])

Parameters:

CSP

Required. The handle to the CSP which is receive the imported key.

BlobType

Required. A parameter specifying the type of key BLOB to be imported. Values are defined in WinCrypt.h

KeyBLOB

Required. Text string containing the KeyBLOB to be imported.

EncryptKey

An optional parameter containing a Key handle for a key to be used to encrypt the exported key so that it may only be encrypted by the destination user. If omitted or invalid, then the value NULL is used.

Flags

An optional parameter specifying the flags to be passed to CryptExportKey. If omitted or invalid then the value 0 is used.

Error

An optional variable in which the error code for the function is returned. It may have the following values:

Error	Meaning
0	Key successfully imported.
1	CSP, BlobType or KeyBLOB parameters invalid.
X	Any other value is an error from CryptImportKey.

Comments: The new key is returned as a Key handle. If an error occurs, the return value is invalid.

Example:

```
[
  Constant PUBLICKEYBLOB = 0x6;
  Key3;
]
Init [
  If 1 Main;
  [
    { Import the public key }
    Key3 = ImportKey(CSP, PUBLICKEYBLOB, PubKey1, Key2);
  ]
]
```

In

Description: Read I/O Byte. This function returns the byte read from an I/O port.

Returns: Byte

Usage: Script or steady state.

Function Groups: Memory I/O, Stream and Socket

Related to: InWord | Out | OutWord

Format: In(Port)

Parameters:

Port

Required. Any numeric expression, which specifies which I/O port to read. Port must be in the range 0 to 65535

Comments: This function requires that the VTSIO driver be installed. Please refer to the topic, Communicating Directly With Hardware for more details.
If In is used in a statement or an action trigger, it will be evaluated at a very fast rate. In is a high priority function, and should be used sparingly to avoid reducing overall system performance.

Example:

```
reg = In(0x300);
```

This reads a byte from CPU input port 300 hex. If this statement appears in a script the read takes place when the script is executed. If this statement appears in steady state, it is updated very rapidly as a high-priority function.

InsertArrayItem

Description: Insert Array Item. This function inserts an element into a dynamically allocated array and returns the modified array.

Returns: Array

Usage: Script Only.

Function Groups: Array

Related to: DeleteArrayItem | New

Format: InsertArrayItem(Array [, Index, Value])

Parameters:

Array

Required. Any variable whose value is invalid or con-

tains a dynamically allocated array (one created via a New function call). This should be a single dimension array or unexpected results may occur.

Index

An optional parameter that is any numeric expression for the index at which to insert a new element. If this value is invalid, the new element will be inserted at the end of the array.

Value

An optional parameter that is any expression for the value to assign to the new array element.

Comments: This function supersedes the System Library's InsertListItem.
This function is intended for use on dynamically allocated arrays, that is, arrays that have been created via the New function. If used with an array that has been statically declared, unless otherwise specified in the Array parameter, the first element of the array will be used, and a dynamically allocated array will be created/added to in this element.

Examples:

```
If 1 Next;
[
  Data = InsertArrayItem(Data { Array to use },
                         Invalid { Insert at end },
                         32 { Value of element });
 Names = InsertArrayItem(Names { Array to use });
]
```

The first statement inserts an element with a value of 32 at the end of array Data, while the second statement inserts an invalid element at the end of array Names.

Instance

Description: Limit Module Instances. This function limits the number of

fixed module instances allowed to run simultaneously and returns the old limit

Returns: Numeric

Usage: Script Only.

Function Groups: Basic Module

Related to: NumInstances | CalledInstances | GetInstance

Format: Instance(Module, Count)

Parameters:

Module

Required. Any text expression that specifies the fixed module to limit; it must be in the current scope – scope resolution operators (\ ) are not permitted.
If the module does not exist, nothing is done. If the module is not a fixed module, the return value is -1.

Count

Required. Any numeric expression that specifies the number of fixed module instances allowed to run simultaneously.
If Count is in the range 1 to 2,147,483,647, the new limit is set and the old limit is returned.
If Count is 0, the limit is not changed, but the current limit is returned.
Count is ignored if the module is not a queued module.

Comments: This function is recommended for experienced users only and is not needed for most applications. If this function is not used, the number of concurrent instances for a given fixed module defaults to 1.

Note: This limit is the number of concurrent instances allowed for each parent instance. Each parent module instance has a separate Count for

each of its fixed modules, which is separate from other instances of itself.

Example:

```
If 1 Main;
[
  mtrs = Instance("Motor", 5);
]
```

Assume that this statement is found in a module called Feeder, which has as its child a module called Motor. Execution of the script will cause mtrs to be assigned a value of 1, and to limit the number of instances of Motors to 5 (note that it is necessary for Motor to be a member or an ancestor of Feeder). All instances of module Feeder will be affected; that is to say, each one will be allowed to have up to 5 running instances of Motor.

Int

Description: Integer Portion of Number. This function returns the portion of a number before the decimal point.

Returns: Integer

Usage: Script or steady state.

Function Groups: Rounding Math

Related to: Ceil | Step

Format: Int(X)

Parameters:

X

Required. Any numeric expression. Normally this is a floating point value.

Examples:

```
a = Int(1.00);
b = Int(1.12);
c = Int(1.99);
```

```
d = Int(2.00);
e = Int(-1.00);
f = Int(-1.9);
```

The variables a, b, c, d, e and f will have values of 1, 1, 1, 2, -1 and -2 respectively.

```
x1 = 3.4;
y1 = 4.7;
x2 = Int(x1 + 0.5);
y2 = Int(y1 + 0.5);
```

The value of x2 and y2 in the above example will be 3 and 5 respectively. Notice how adding 0.5 to each value causes the Int function to perform mathematical rounding rather than truncation.

Comments:
The Int function is sometimes referred to as "floor."

Intgr

Description: Time Integral. This function returns the time integral of a value.

Returns: Numeric

Usage: Steady State only.

Function Groups: Generic Math

Related to: Deriv | PID

Format: Intgr(Value, Time)

Parameters:

Value

Required. Any numeric expression giving the value to integrate with respect to time.

Time

Required. Any numeric expression giving the maximum time in seconds between integral function updates.

Comments: The integral function takes the Value parameter, multiplies by the elapsed time and adds it to the accumulated value so far. If the Value changes from invalid to valid, the new valid result will start at zero. This function is the inverse of Deriv.

The time parameter is necessary because of VTScada's evaluation method of not doing any calculations unless necessitated by a change in a parameter. This means that if Value remains unchanged, the Intgr function will be re-calculated after the time interval specified by the Time parameter.

This function is often used in control functions such as PID loops where it makes up the "I" in the "PID."

The Intgr function is reset when it appears in a true action trigger, when a state starts, or when it appears in a function which resets its parameters (e.g. Latch, Toggle, and Save). When Intgr is reset, the integral starts from zero again.

Examples:

```
speed = Intgr(accel, 0.1);
```

This computes speed as the time integral of accel. The integral starts when the state containing this function starts, and stops when the state stops. The integral is updated every 0.1 seconds.

```
runTime = Intgr(motorOn, 0.25);
```

This integrates a constant 1 while the motor is running, 0 otherwise. This computes the running time of the motor, accurate to 0.25 seconds.

Invalid

Description: Return Invalid Value. This function always returns an invalid value.

Returns: Invalid

Function Groups:	Logic Control, Variable
Usage:	Script or steady state.
Related to:	Valid
Format:	Invalid()
Parameters:	None
Comments:	This function is useful to invalidate data that are found to be incorrect, or to disable statements or functions that will not execute with invalid parameters.

Examples:

```
x = Valid(Invalid());
```

This will set x to 0.
Another example of this function's usefulness follows:

```
Main [
  If !Valid(a);
  [
    a = Scale(Rand(), 0, 1, -1000, 1000);
    ...
  ]
  If Valid(a);
  [
    ...
    a = Invalid();
  ]
]
```

By toggling a between a valid and an invalid value, you can ensure that the two scripts in Main will take turns executing; it is not possible for a race condition to occur.

InWord

Description:	Read I/O Word. This function reads a 16 bit unsigned word from an I/O port.
Returns:	Word
Usage:	Script or steady state.

Function Groups: Memory I/O

Related to: In | Out | OutWord

Format: InWord(Port)

Parameters:

Port

Required. Any numeric expression that gives the I/O address. Port must be in the range 0 to 65535.

Comments: This function requires that the VTSIO driver be installed. Please refer to the topic, Communicating Directly With Hardware for more details.
This is a high priority function. If InWord is used in a state, it will be evaluated at a very fast rate. InWord should be used sparingly to avoid reducing system performance.

Examples:

```
reg = InWord(0x300);
```

This reads reg as a 16 bit unsigned word from input port 300 hex. If this statement appears in a script, it is executed when the script is executed (when the action is triggered); if it appears in a state, it will be updated rapidly as a high priority function.

IPAddressList

Description: Displays a list of IP address which can be added to or removed from.

Returns: Text

Usage: Steady State only.

Function Groups: Graphics, String and Buffer

Related to: PIPAddressList

Format: \IPAddressList(X1, Y1, X2, Y2, AddressList, BaseFocusID[, Title, DrawBevel, AlignTitle, ListBGndColor, Over-

layCallback]);

Parameters:

X1

Required. Any numeric expression giving the X coordinate on the screen of one side of the object and its label. The smaller of X1 and X2 will always be to the left

Y1

Required. Any numeric expression giving the Y coordinate on the screen of either the top or bottom of the object. The smaller of Y1 and Y2 will always be the top.

X2

Required. Any numeric expression giving the X coordinate on the screen of the side of the object and its label opposite to X1.

Y2

Required. Any numeric expression giving the Y coordinate on the screen of the top or bottom of the object, whichever is the opposite to Y1.

AddressList

Required. The array of IP addresses to be displayed / the array of IP addresses returned.

BaseFocusID

Required. Reserves 10 focus ID values beginning at the value provided. Set to zero to disable user input.

Title

Optional. Any text expression to use as the title for the field. No default value

DrawBevel

Optional. Any logical expression. If TRUE, a bevel is

drawn around the graphic. Defaults to FALSE.

AlignTitle

Optional. Any logical value. If TRUE then title affects alignment

ListBGndColor

Optional. Any numeric expression setting a background color for the list. No default value.

OverlayCallback

Optional module value. Called from the listbox. No default value.

Comments: Control is read-only if BaseFID is zero.
Addresses are validated by a utility function in SocketServerManager before being added to the list.
Controls are available for removing items and reordering.
A double-click on a list item will copy it into the edit field.
The resulting list is stored in the AddressList array, which also serves as the initial display on initialization. Note that addresses are only validated when the add button is pressed.

Example:

The following example is taken from the PIPAddressList code. The overlay callback allows the caller to react to user actions such as right-clicks.

```
\IPAddressList(LHS, BTM, RHS, TOP, AddressArray,
               AllowEdit ? ID : 0,
               Title, DrawBevel, AlignTitle, HighlightColor,
               Variable("OverlayCallback") { List overlay graphics
callback });
```

IsActive

(Alarm Manager module)

Note: Use GetAlarmStatus in new code.

Description: Will indicate if an alarm is active. It can be used either as a subroutine or as a called function.

Returns: Numeric

Usage: Script Only.

Function Groups: Alarm

Related to: GetAlarmStatus | IsShelved | IsDisabled | IsUnacked

Format: \AlarmManager\IsActive(AlarmName);

Parameters:

AlarmName

Required. A text expression providing the name of an alarm. Not the alarm object value that was passed to the Register subroutine.

Comments: The IsActive subroutine returns a "1" if the alarm is active; otherwise it returns a "0".

Example:

Within a custom tag...

```
AlarmStatus = \AlarmManager\GetAlarmStatus(Root\UniqueID);
AlarmActive = AlarmStatus\IsActive;
```

IsAppEditable

Description: Returns TRUE if the application can accept changes without being re-started.

Returns: Boolean

Usage: Script or steady state.

Function Groups: Configuration Management

Related to: HasCompilationErrors | HasUndeployedChanges | AppIsRunning | GetAppInstance | GetLoadedAppInstance | GetOEMLayer

Format: \LayerRoot\IsAppEditable() or \IsAppEditable()

Parameters: none

Comments: Note that a steady-state call of this function is not automatically updated with a repository change.
This function can evaluate to false if applying an outstanding set of changes would cause an error or if there are outstanding changes that require a restart to be applied. It will also return FALSE if the working copy is not at the repository tip, or if there is a compilation error (generally due to corruption of the working copy files).
The Layer object can be acquired using GetAppInstance, GetLoadedAppInstance or GetOEMLayer.

Examples:

```
Main [
   CanEditLayer = Layer\IsAppEditable();
   ...
```

```
If WorkingCopyLock Commit;
[
   { Now that we have the WC lock, make sure the Layer still is editable }
   IfElse(Layer\IsAppEditable(),
      WriteINIPropertiesObj = Layer\WriteINIProperties(ConfigData, TRUE);
   { Else }
      ForceState("ReleaseLock"); { Abort if app not editable }
   );
]
```

IsChild

Description: Identify Child Module. This function returns an indication of whether one module is a child module of another.

Returns: Boolean

Usage: Script Only.

Function Groups: Basic Module

Related to:

Format: IsChild(Child, Parent)

Parameters:

Child

Required. Any expression that returns an object or module value.

Parent

Required. Any expression that returns an object or module value.

Comments: This function returns true if Child is a descendant module of Parent, false if it isn't, and invalid if either, or both arguments are invalid or not module or object values.

Example:

```
If ! Valid(isMember);
[
  isMember = IsChild(read1 { Read module object value },
  driver { I/O driver module object value });
]
```

The variable isMember will be set to true if the object read1 is a child of (defined within) the object driver.

IsClient

(RPC Manager Library)

Description: Is Client of a Service. This subroutine returns an indication of whether or not a particular workstation is a client connected to a service. Returns 1 for the specified service if the specified machine is a client to the machine on which the IsClient() call is made.

Returns: Boolean

Usage: Script Only.

Function Groups: Network

Related to: ConnectToMachine | DisconnectFromMachine | GetServer | GetServersListed | GetStatus | IsPotentialServer | IsPrimaryServer | Register (RPC Manager) | Send | SetRemoteValue

Format: \RPCManager\IsClient(ServiceName, Workstation [, OptGUID])

Parameters:

ServiceName

Required. Any text expression giving the name by which the service is known.

Workstation

Required. Any text expression giving the name or IP address by which the workstation is known to the RPC Manager.

OptGUID

Any optional parameter that provides the GUID for the application in which the service instance is located. The default is the application to which the caller belongs.

Comments: This subroutine is a member of the RPC Manager's Library, and must therefore be prefaced by \RPCManager\, as shown in the "Format" section. If the application you are developing is a script application, the subroutine call must be prefaced by System\RPCManager\, and the System variable must be declared in AppRoot.src.

The return value from this subroutine is a logical value, if true (1) the workstation is a client connected to the service, if false (0) it is not.
If the 16-byte binary format of the GUID is not known, the GetGUID function may be used to obtain it.

Example:

```
If 1 Main;
[
  IfThen(\RPCManager\IsClient("ModemManager",
  "TestMachine"),
  ...
```

```
  );
]
```

Related Information:

You may also refer to "RPC Manager Service" for a listing of Service Control Methods, RPC Methods, and Deprecated RPC Methods.

IsEqual

Description: Will return TRUE if the parameter values are equivalent, or if both are invalid.

Returns: Boolean

Usage: Script or steady state.

Function Groups: Variable

Related to: Valid

Format: IsEqual(Parm1, Parm2);

Parameters:

Parm1

Required. Any value, to be compared to parameter 2.

Parm2

Required. Any value, to be compared to parameter 1.

Comments: This function is equivalent to the longer expression: !Valid(Parm1) == !Valid(Parm2) OR Parm1 == Parm2
If the parameters are text or numeric, they will be compared directly. If the parameters are expressions, the strings that they evaluate to will be compared. If object values, the Name variables will be used, thus allowing tag to tag comparisons. Other value types will be turned into strings before the comparison.

Examples:

```
If !IsEqual(Color, ChosenColor);
[
   Color = ChosenColor;
   EnableColorSelect = 0;
]
```

IsDictionary

Description: A synonym for HasMetadata. Tests whether the parameter is a dictionary.

Returns: Boolean

Usage: Script Only.

Function Groups: Dictionary, Variable

Related to: HasMetaData | Dictionary

Format: IsDictionary(Value);

Parameters:

Value

Required. Any value to be tested.

Comments: Any value can be entered as the lone, required, parameter. The result is TRUE if the value is of type Dictionary, regardless of the presence of a root value. Otherwise, a FALSE will always be returned. An invalid parameter will cause a response of FALSE. IsDictionary cannot have an invalid outcome.

IsDisabled

(Alarm Manager module)

Note: Use GetAlarmStatus in new code.

Description: Will indicate if an alarm is disabled. It can be used either as a subroutine or as a function.

Returns: Numeric

Usage: Script Only.

Function Groups: Alarm

Related to: GetAlarmStatus | IsActive | IsShelved | IsUnacked

Format: \AlarmManager\IsDisabled (AlarmName);

Parameters:

AlarmName

Required. A text expression providing the name of an alarm. Not the alarm object value that was passed to the Register subroutine.

Comments: The IsDisabled subroutine returns a "1" if the alarm is disabled; otherwise it returns a "0".

Example:

Within a custom tag...

```
AlarmStatus = \AlarmManager\GetAlarmStatus(Root\UniqueID);
AlarmDisabled = AlarmStatus\IsDisabled;
```

IsLoggedOn

Security Manager Module

Description: Returns TRUE if the calling user is logged on, else FALSE.

Returns: Boolean

Usage: Script or steady state.

Related to: GetAccountID | GetAccountInfo | GetFullName | GetGroupName | GetUserName | IsSecured | IsSuspended | SecurityCheck | UIErrorToText

Format: \SecurityManager\IsLoggedOn()

Parameters: None

Comments: None

IsMatch

(RPC Manager Library)

Description: Determines whether two names or IPs indicate the same workstation. This subroutine returns a "1" if the two names or IPs (any combination) refer to the same workstation.

Returns: Boolean

Function Groups: Network

Usage: Steady State only.

Related to:

Format: \RPCManager\IsMatch(Name1, Name2);

Parameters:

Name1

Required. Any of the names or IPs by which the first workstation is known to the RPC Manager.

Name2

Required. Any of the names or IPs by which the second workstation is known to the RPC Manager.

Comments: This subroutine is a member of the RPC Manager's Library, and must therefore be prefaced by \RPCManager\, as shown in the "Format" section. If the application you are developing is a script application, the subroutine call must be prefaced by System\RPCManager\, and the System variable must be declared in AppRoot.src.
The return value from this function will be a "1" if the two names or IPs (any combination) refer to the same workstation.

IsOnLocalBranch

Description: Returns TRUE if the local machine is maintaining changes that have not been deployed within the repository.

Returns: Boolean

Usage: Script or steady state.

Function Groups: Configuration Management

Related to: HasCompilationErrors | IsAppEditable | IsRunOnly | HasUndeployedChanges | GetAppInstance | GetLoadedAppInstance | GetOEMLayer

Format: LayerModule\IsOnLocalBranch()

Parameters: None

Comments: Also defined as the repository tip being on the local branch.
Useful for determining whether there is anything that can be deployed or reverted.
Note that a steady-state call to this function will not be updated automatically with a repository change.
The Layer object can be acquired using GetAppInstance, GetLoadedAppInstance or GetOEMLayer.

IsPotentialServer

(RPC Manager Library)

Description: Is Potential Server for a Service. This subroutine returns an indication of whether or not the local workstation is a potential server for a service. Returns "1" if the local workstation can be a server for the specified service. IsPotentialServer should not be called in steady state.

Returns: Boolean

Usage: Script Only.

Function Groups: Network

Related to: ConnectToMachine | DisconnectFromMachine | GetServer | GetServersListed | GetStatus | IsClient | IsPrimaryServer | Register (RPC Manager) | Send | SetRemoteValue

Format: \RPCManager\IsPotentialServer(ServiceName [, OptGUID])

Parameters:

ServiceName

Required. The name by which the service is known.

OptGUID

Any optional parameter that provides the GUID of the application in which the service instance is located. The default is the application to which the caller belongs.

Comments: This subroutine is a member of the RPC Manager's Library, and must therefore be prefaced by \RPCManager\, as shown in the "Format" section. If the application you are developing is a script application, the subroutine call must be prefaced by System\RPCManager\, and the System variable must be declared in AppRoot.src.
This function returns Invalid if the local service instance is not running.
If the 16-byte binary format of the GUID is not known, the GetGUID function may be used to obtain it.

Example:

```
If 1 Main;
[
  IfThen(\RPCManager\IsPotentialServer("ModemManager"),
  ...
  );
]
```

Related Functions:

Refer also to "RPC Manager Service" for a listing of Service Control Methods, RPC Methods, and Deprecated RPC Methods.

IsPrimaryServer

(RPC Manager Library)

Description: Is Primary Server Active for a Service. This module returns an indication of whether or not the active server for a service is the primary server. Returns "1" if the local workstation is the current server for the specified service.

Returns: Integer

Usage: Steady State only.

Function Groups: Network

Related to: ConnectToMachine | DisconnectFromMachine | GetServer | GetServersListed | GetStatus | IsClient | IsPotentialServer | Register (RPC Manager) | Send | SetRemoteValue

Format: \RPCManager\IsPrimaryServer(ServiceName [, OptGUID])

Parameters:

ServiceName

Required. The name by which the service is known.

OptGUID

Any optional parameter that provides the GUID of the application in which the service instance is known. The default is the application to which the caller belongs.

Comments: This module is a member of the RPC Manager's Library, and must therefore be prefaced by \RPCManager\, as shown in the "Format" section. If the application you are developing is a script application, the module call must be prefaced by System\RPCManager\, and the System variable must be declared in AppRoot.src.
The return value from this module is a logical value, if true (1) the primary server is the active server for the service, if false (0) a backup or no server is active is active.
If the 16-byte binary format of the GUID is not known, the GetGUID function may be used to obtain it.

Example:

```
If \RPCManager\IsPrimaryServer("ModemManager") Main;
[
  ...
]
```

Related Functions:

Refer also to "RPC Manager Service" for a listing of Service Control Methods, RPC Methods, and Deprecated RPC Methods.

IsRunning

Description: Check if a Program is running. This function returns an indication of whether a certain program is running on the same computer.

Returns: Boolean

Usage: Steady State only.

Function Groups: Software and Hardware

Related to: DDE | Spawn

Format: IsRunning(Program)

Parameters:

Program

Required. Any text expression giving the program name to test (without the .EXE extension).

Comments: This function returns 1 if the specified program is running on the same computer.
If running under 32-bit, this statement will only be valid if the value of Program refers to another 32-bit process.

Example:

```
If ! Valid(running);
[
  running = IsRunning("Excel");
]
```

After executing this statement, the variable running will have a value of 1 if the program Microsoft™ Excel is running and a 0 if not.

IsRunOnly

Description: Returns TRUE if the application is a run-file-only app, according to the WC contents.

Returns: Boolean

Usage: Script Only.

Function Groups: Configuration Management

Related to: IsRunning | GetAppInstance | GetLoadedAppInstance | GetOEMLayer

Format: \LayerModule\IsRunOnly()

Parameters: none

Comments: The application is defined as run-only if it does not contain source files.
This function should only be launched as a subroutine.

IsSecured

Security Manager Module

Description: Returns TRUE if the application has any user accounts defined, else FALSE.

Returns: Boolean

Usage: Script or steady state.

Related to: GetAccountID | GetAccountInfo | GetFullName | GetGroupName | GetUserName | IsLoggedOn | IsSuspended | SecurityCheck | UIErrorToText

Format: \SecurityManager\IsSecured()

Parameters: None

Comments: An application can be unsecured, yet have role accounts configured. An application is only considered secured if one or more user accounts exist.

IsServiceReady

(RPC Manager Library)

Description: Is Primary Server Active for a Service. Only available in VTS 6. This module returns an indication of whether or not the

specified server is in synchronization with the server instance. Returns "1" if the local instance is in synchronization with the server instance.

Returns: Boolean

Usage: Steady State only.

Function Groups: Network

Related to: ConnectToMachine | DisconnectFromMachine | GetServer | GetServersListed | GetStatus | IsClient | IsPrimaryServer | IsPotentialServer | Register (RPC Manager) | Send | SetRemoteValue

Format: \RPCManager\IsServiceReady(ServiceName [, OptGUID])

Parameters:

ServiceName

Required. The name by which the service is known.

OptGUID

Required. Any optional parameter that provides the GUID of the application in which the service instance is known. The default is the application to which the caller belongs.

Comments: This module is a member of the RPC Manager's Library, and must therefore be prefaced by \RPCManager\, as shown in the "Format" section. If the application you are developing is a script application, the module call must be prefaced by System\RPCManager\, and the System variable must be declared in AppRoot.src.
If the 16-byte binary format of the GUID is not known, the GetGUID function may be used to obtain it.

Related Functions:

Refer also to "RPC Manager Service" for a listing of Service Control Methods, RPC Methods, and Deprecated RPC Methods.

IsShelved

(Alarm Manager module)

Note: Use GetAlarmStatus in new code.

Description: Will indicate if an alarm is shelved. It can be used either as a subroutine or as a called function.

Returns: Boolean

Usage: Script Only.

Function Groups: Alarm

Related to: GetAlarmStatus | IsActive | IsDisabled | IsUnacked

Format: \AlarmManager\IsShelved(AlarmName);

Parameters:

AlarmName

Required. A text expression providing the name of an alarm.

Comments: The IsShelved subroutine returns a "1" if the alarm is active; otherwise it returns a "0".

Example:

Within a custom tag...

```
AlarmStatus = \AlarmManager\GetAlarmStatus(Root\UniqueID);
AlarmShelved = AlarmStatus\IsShelved;
```

IsSuspended

Security Manager Module

Description Returns TRUE if the user's account is suspended, else FALSE.

Returns Boolean

Usage Script or steady state.

Related to: GetAccountID | GetAccountInfo | GetFullName |

GetGroupName | GetUserName | IsLoggedOn | IsSecured | SectionControl | UIErrorToText

Format: \SecurityManager\IsSuspended()

Parameters None

Comments A user's account will become suspended when a user tries to log on, but a password change is required. IsSuspended () will return TRUE until the user closes the password change dialog.

Related Functions:

IsUnacked

(Alarm Manager module)

Note: Use GetAlarmStatus in new code.

Description Will indicate if an alarm is unacknowledged. It can be used either as a subroutine or as a function.

Returns Numeric

Usage Script Only.

Function Groups Alarm

Related to: GetAlarmStatus | IsActive | IsDisabled | IsShelved

Format: \AlarmManager\IsUnacked(AlarmName);

Parameters

AlarmName

Required. A text expression providing the name of an alarm. Not the alarm object value that was passed to the Register subroutine.

Comments The IsUnacked subroutine returns a "1" if the alarm is unacknowledged; otherwise it returns a "0".

Related Functions:

IsVICSession

Description Returns TRUE to indicate that a call is being made from a VTScada Internet Client session.

Returns Boolean

Usage Script Only.

Function Groups VTScada Internet Client

Related to:

Format: IsVICSession([IsAnywhere])

Parameters

IsAnwhere

Optional Boolean. When set TRUE, makes IsVICSession only return true if the session is an Anywhere client session. When FALSE (0), the function returns true if the client is either a VIC or an Anywhere client. Defaults to 0.

Comments Certain modules need to know whether they are being executed from a VTScada workstation or a VIC, and will adjust their behavior as needed. IsVICSession provides a quick test to determine this.

K Functions

The sections that follow identify all VTScada functions beginning with "K".

KeyCount

Description: Returns the number of keys pressed since the state became active, either edited or non-edited.

Returns: Numeric

Usage: Steady State only.

Function Groups: Keyboard

Related to: KeyFake | Keys | MatchKeys

Format: KeyCount(Option)

Parameters:

Option

Any logical expression. If true, the number of non-edited keystrokes is returned; if false, the edited keystrokes is returned.

Comments: Edited keystrokes are printable characters and the enter key. Non-edited keystrokes are all keystrokes, including printable characters, enter key, special keys, and function keys.
When the Backspace key is pressed, the key code for backspace is entered as a non-edited keystroke, and the previous edited keystroke is removed. That is, when Backspace is pressed the number or non-edited keystrokes increments, and the number of edited keystrokes decrements (and possibly goes negative).

Examples:

```
If KeyCount(1) MainMenu;
```

This action simply waits for any key to be pressed, and changes to another state. Another way of using this function would be:

```
If KeyCount(1) > keysChecked;
```

This checks that the number of non-edited keys pressed is greater than the value of keysChecked. This is the beginning of a loop which would process keystrokes one at a time (such as a text editor).

KeyFake

Description Places a string of characters in a window's keyboard buf-

fer.

Returns Nothing

Usage Script Only.

Function Groups Keyboard, String and Buffer

Related to: KeyCount | Keys | MatchKeys

Format: KeyFake(Object, String)

Parameters

Object

Any expression that returns an object value that defines a window.

String

Any text expression that gives the characters to place in the window's keyboard buffer.

Examples:

```
If 1 Main;
[
  KeyFake(myWindow, "Hello World");
]
```

Keys

Description Returns the most recently pressed keys, optionally flagging virtual key codes.

Returns Text

Usage Script or steady state.

Function Groups Keyboard, String and Buffer

Related to: KeyFake | KeyCount | MatchKeys

Format: Keys(N, Option)

Parameters

N

Any numeric expression giving the number of keys to get.

Option

Any logical expression indicating whether edited mode or non-edited mode should be used. Set to TRUE for the non-edited mode.

Comments In the non-edited mode, function keys and others that return a virtual key code will have a 0xFD (253) pre-pended to the return value. (See Microsoft's online MSDN reference for a list of virtual key codes.) Because of this, the return value may contain more bytes than specified by the parameter, N.

Examples:

```
nonEditPressed = Keys(5, 1);
```

This sets nonEditPressed to a text string containing the key codes for the last five keys pressed, in order from oldest to most recently pressed.
In Edited Mode, the keys [a][b][c][up-arrow][d] results in the buffer, 61 62 63 64.
In Non-edited mode, [a][b][c][up-arrow][d] results in 61 62 63 FD 48 64.

L Functions

The sections that follow identify all VTScada functions beginning with "L".

LastSelected

Description: Returns the most recently selected graphics statement.

Returns: Text

Usage: Script Only.

Function Groups: Compilation and On-Line Modifications, Graphics

Related to:

Format: LastSelected(Object)

Parameters:

Object

Required. Any object value expression that defines the window where the selected items are found.

Comments: None

Latch

Description Latch On or Off. This function allows a transient change of a variable to be captured. Its return value is determined by the rules listed in the comments section.

Returns Boolean

Usage Steady State only. See: Rules for Usage.

Function Groups Variable

Related to: Toggle | MatchKeys | TimeOut | Intgr | RTimeOut

Format: Latch(Set, Reset)

Parameters

Set

Required. Any numeric expression. When true (i.e. not equal to 0), the latch is set.

Reset

Required. Any numeric expression. When true (i.e. not equal to 0), the latch is reset.

Comments This function starts in a state with its return value being a valid 0 (false). The change in values is governed by the following rules

Set	Reset	Old Value	New Value
FALSE	FALSE	FALSE	FALSE
FALSE	FALSE	TRUE	TRUE
Either	TRUE	Either	FALSE
TRUE	FALSE	Either	TRUE
Invalid	TRUE	Either	FALSE
Either	Invalid	Either	Invalid

If both the Set and Reset are false, the latch value remains unchanged. Note that a true Reset overrides a true Set.
This function resets its parameters after they evaluate to true. This is significant only for functions which can be reset, such as MatchKeys, TimeOut, Intgr and RTimeOut.

Example:

```
motorOn = Latch(MatchKeys(1,"1") { Set the var with a "1" },
                MatchKeys(1,"0") { Reset the var with a "0" });
```

This controls the variable motorOn from the keyboard. MotorOn initially starts as 0. When 1 is pressed on the keyboard, the latch is set and motorOn becomes 1. When 0 is pressed on the keyboard, the latch is reset and motorOn becomes 0.

Related Information:

See: "Latching and Resetting Functions" in the VTScada Programmer's Guide

Launch

Description Runs a module instance and returns a pointer to it.

Returns Varies – see comments

Usage Script Only.

Function Groups Basic Module

Related to: Call | FindVariable | LoadModule | Return | Slay | Thread

Format: Launch(Launchee, Parent, Caller[, P1, P2, ...])

Parameters

Launchee

Required. A module pointer or a text expression giving the module to run.
If this parameter is a module value that has been returned from a LoadModule or FindVariable statement, that module will be launched.
If this parameter is type text it may either designate the module by its name, if it is in scope, or it may give the name of a variable that contains the module value of the module to launch.

Parent

Required. The object value of the module where the Launchee is to resolve its global variable references.
If a valid non-object value is supplied the Launchee will resolve its global variable references to the scope defined by the first parameter.
If this is invalid, the module will still run, but global references will be invalid.

Caller

Required. The object value of the window to draw in.
This specifies the module instance where the Launchee acts as if it were called from.
If this is invalid, the module will still run but will not stop without a Slay. A Return in the case of a subroutine.
If it is valid, the module will stop when the Caller module instance stops, when a Slay is executed upon it, or (subroutine only) when it calls Return .

P1, P2, ...

Optional. Are any expressions that will be supplied as parameters to the launched module.

Comments

This function returns an object value of the newly started module. In general, variables that are listed in the final parameter spots are passed to the module as a value only. This means that if the launched module instance changes the value of one of the parameters, its value will not change outside of the scope of the module. If there are variables external to the module that the module itself will be required to alter, it is best to make them within the scope of Parent.

If a launched module contains a Return statement it is considered to be a sub-routine, whether it is explicitly or implicitly launched. This is true even if the Return statement is in a state that does not get executed. In that case, the script that launches the sub-routine will stop its execution indefinitely, waiting for the sub-routine to return a value.

If a sub-routine contains the statement

```
x = Launch("myMod", …);
```

x will be set to the value returned by the sub-routine's Return statement upon completion. Prior to execution of the Return statement, x will be invalid, unlike a module that is not a sub-routine, which would set x to the object value of the launched module.

A common syntactical problem with the use of the Launch function will result in two instances of a module running. The following example shows the improper syntax

```
Launch(X, Self, Self);
```

(assuming that "X" is declared as a module.) The problem here is the first parameter. As written above, the code will launch a module called "X", and use the return value from the launch (an object value) as the first parameter for Launch . This will result in two copies of the "X" module running. The correct syntax is:

```
Launch("X", Self, Self);
```

Example:

```
If ! Valid(modPtr);
[
  modPtr = Launch(FindVariable("DataLog", Self(), 0, 1),
                  Self(), Self(),  { Parent and caller }
                  timeSpan, fileSave { Parameters });
]
```

This launches one instance of the module DataLog. The current module is its parent and caller. The two variables timeSpan and fileSave are passed as parameters to the module.

LayerInUse

Description: Returns true if the application is running, or if there are any applications that depend on this layer, running or not.

Returns: Boolean

Usage: Script Only.

Function Groups: Configuration Management

Related to:

Format: LayerRoot\LayerInUse()

Parameters: None

Comments: If this function returns FALSE, the layer is not in use and may be removed from the VAM.

Examples:

LayerRoot\Stop

Description: Stop the application designated by LayerRoot

Returns: Nothing

Usage: Script Only.

Function Groups: Configuration Management

Related to: Start |

Format: LayerRoot\Stop

Parameters:

IsRestart

Optional Boolean. Set TRUE if the application is to be re-started.

Comments: For a VTScada application, most of the work is done in Start\VTSThread. Once the app is stopped, AppStopMonObj does some post-stop clean-up. The work is done there rather than here as a script application often stops just by slaying itself, and AppStopMonitor picks up on both situations.
This module launches a worker module into the Layer so that the operation is not interrupted by this module's caller being slain.

Examples:

Building on the example in Start(), the following will stop the application started in that example.

```
CompLayer\Stop();
```

Related Functions:

Limit

Description Set Value Minimum and Maximum. This function returns a

value that is limited both on the high and low ranges.

Returns Numeric

Usage Script or steady state.

Function Groups Rounding Math

Related to: Max | Min

Format: Limit(X, Low, High)

Parameters

X

Required. The variable whose numeric value will be limited.

Low

Required. Any numeric expression giving the lower limit for the value X.

High

Required. Any numeric expression giving the upper limit for the value X.

Comments This function performs the same operation as Min(High, Max(Low, X)) but is simpler to write and requires less memory. If the value exceeds one of the limits, the function returns that limit. Otherwise, the value is returned unchanged. If Low is greater than High, the value for High is returned. If any of the parameters is invalid, the function returns invalid.

Examples:

```
a = Limit(123.4, 0, 100);
b = Limit(-2.6, 0, 100);
c = Limit(36.7, 0, 100);
d = Limit(36.7, Invalid, 100);
```

The values for a, b, c, and d are 100, 0, 36.7 and invalid, respectively.

Line

Note: Deprecated. Do not use in new code.

Description: Draws a line on the screen that may consist of multiple segments.

Returns: Numeric

Usage: Steady State only.

Function Groups: Graphics

Related to: GUIPolygon | Pipe | PlotXY | ZLine

Format: Line(Style, Width, Color, Curvature, X1, Y1, X2, Y2, ...)

Parameters:

Style

Required. Any numeric expression giving the Line Types. Valid line styles are from 1 to 5 inclusive. A line style of 1 is a solid line

Width

Required. Any numeric expression giving the width of the line in units of X screen coordinates. The width is always rounded to result in an odd number of pixels on the screen. The minimum width displayed will be 1 pixel.

Color

Required. Any numeric expression giving the VTScada Color Palette of the line.

Curvature

Required. Any numeric expression giving the radius of curvature of the corners for the line. This is specified in units of X screen coordinates. If the number of endpoints is 2, Curvature is ignored.

X1, Y1, X2, Y2, ...

Required. Any numeric expressions giving the screen

coordinates of the line endpoints.

Comments: This statement has been superseded by the GUIPolygon and ZLine statements and is maintained for backwards compatibility only.
The radius of curvature of the line corners is the radius of the arc that joins the line endpoints. A Curvature of 0 results in sharp (square) line corners. Larger Curvature numbers result in greater rounding of the line corners.
As of version 11, this is now drawn in the same z-order as other graphics, making it similar to the z-graphics functions.

> Note: Within an Anywhere Client session, this function does nothing.

Example:

```
Line(1 { Solid line style },
     1 { Line width is 1 pixel },
     10 { Light green },
     0 { Curvature not applicable (2 endpoints) },
     0, 0 { Coordinates of first end },
     XLoc(), YLoc() { Second end follows mouse });
```

This statement will draw a light green solid line with one end anchored at (0, 0) and the second end following the movements of the mouse.

LinearIndicator

(Meter Parts Library)

Description: Will draw a linear type indicator. A linear indicator can be drawn in 3 different ways. Scaled from min to current position, cropped from min to current position or as a line that moves to the current position. This function must be called inside a GUITransform in order to work properly.

Returns: Image handle

Usage: Steady State only.

Function Groups: Graphics

Related to:

Format: \MeterParts\LinearIndicator(DataSource, IndicatorImage, Orientation, DrawMode, IndicatorRToL, Hue, Saturation, Brightness, Transparency, Contrast, ColorizeHue, Col-orizeIntensity, UseTagScaling, MinScaleValue, MaxScaleValue, DampenIndicator, NumberOfSteps)

Parameters:

DataSource

Required. A Tag name, constant or expression that provides the value to show.

IndicatorImage

Required. The full path to the name of an image file to use as the indicator. Typically this is an image of a needle.

Orientation

A flag indicating the orientation of the Indicator. Set to 0 for Horizontal and 1 for Vertical. The default is 0 (Horizontal).

DrawMode

Controls how the indicator is drawn. Details of the 3 modes are as follows

DrawMode	Meaning
0	Scaled. This mode will scale the Indicator image from the 0 position to the position that represents the current value of the DataSource.
1	Cropped. This mode will scale the indicator image from the 0 position to the full position and then crop it to the position that represents the current value of the DataSource.
2	Moving This mode will keep the Indicator image at a constant size and simply move it to the position that represents the current value of the DataSource.

IndicatorRToL

A flag that changes the location of the 0 position. If set to true, the 0 position is on the right side or the top in vertical orientation.

If set to false and the 0 position is on the left side or the bottom in vertical orientation. The default is false.

Hue

The hue translation to perform on the Indicator image. This enables you to change the color.

The image must have color in it already in order to perform a hue translation. If there is no color to start with, then changing this value does nothing. You can add color by setting a value for the ColorizeHue parameter,

described later.

The default is 0, indicating that no hue translation is done and the indicator is in its native color.

Saturation

The amount of saturation of the colors in the indicator image. A value of 0 will make the image black and white (no color saturation). A value of 2 produces a brightly colored (saturated) indicator. The default is 1 which corresponds to the native saturation of the indicator image.

Brightness

An adjustment of the brightness of the indicator image. Higher numbers produce a brighter indicator image. A 0 produces a black image. The default is 1 which corresponds to the native brightness of the indicator image.

Transparency

An adjustment of the opacity of the indicator where 1 means 100% opacity and 0 means %100 transparent. The default is 1.

Contrast

An adjustment of the contrast of the colors in the indicator image. A value of 0 produces a flat looking image and a value of 2 gives a high contrast image. The default is 1 which corresponds to the native contrast of the indicator image.

ColorizeHue

A value that works in conjunction with ColorizeIntensity. This is the hue of the color that is introduced by colorizing an image. Colorizing an image will introduce color into an image that previously was black and white or grayscale. The default value is 0.

ColorizeIntensity

A value to define how much color to introduce into the image. The default is 0, meaning not to introduce any color at all into the image.

UseTagScaling

A flag that indicates whether or not to use the supplied Tag's scaling values. The default is false.

MinScaleValue

The minimum scale value to use if the UseTagScaling flag is not true. The default is 0.

MaxScaleValue

The maximum scale value to use if the UseTagScaling flag is not true. The default is 100.

DampenIndicator

A flag to indicate whether or not to dampen the indicator movement. Dampened movement creates the effect of animating the indicator. The default is false.

Comments: This function must be called within a GUITransform statement in order for it to work correctly.
The size of the indicator image is scaled with respect to the original size of the image and the size of the transform. If you want a smaller indicator you can simply make a smaller transform.

Example:

```
GUITransform(706, 212, 856, 192,
             1, 1, 1, 1, 1 { Scaling             },
             0, 0          { Movement            },
             1, 0          { Visibility, Reserved },
             0, 0, 0       { Selectability       },
             Variable("Code\MeterParts")\LinearIndicator(Invalid,
      "Bitmaps\Meter Parts\Indicators\Linear\LIndicator2.png", 0,
             2, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 100, 0, 0));
```

LinearLegend

(Meter Parts Library)

Description Draws a legend (i.e. the text labels) for a linear type meter. They are drawn in a line, either horizontally or vertically, with consistent spacing. This function must be called inside a GUITransform in order to work properly.

Returns Image handle

Usage Steady State only.

Function Groups Graphics

Related to:

Format: \MeterParts\LinearLegend(TagName, Orientation, NumLabels, Font, Color, Reserved, UseTagScaling, MinScaleValue, MaxScaleValue)

Parameters

TagName

Required. The name of the Tag to use for scaling. If no tag is specified, then tag scaling cannot be used to automatically obtain the minimum and maximum scale values.

Orientation

A flag indicating the orientation of the legend. Set to 0 for horizontal and 1 for vertical. The default is 0 (Horizontal).

NumLabels

The number of Labels to show. The default is 3.

Font

The name of a Font tag to use for the legend text.

Color

A color index for the color of the legend text. The default is 0 (black).

Reserved n/a

For use at a later time. Should be set to 0.

UseTagScaling

A flag that indicates whether or not to use the supplied tag's scaling values. The default is false.

MinScaleValue

The minimum scale value to use if the UseTagScaling flag is not true. The default is 0.

MaxScaleValue

The maximum scale value to use if the UseTagScaling flag is not true. The default is 100.

Comments This function must be called within a GUITransform statement in order for it to work correctly.
The text should scale with the size of the transform. If it does not, then you might have picked a font that doesn't scale. Some non true-type fonts will not scale.

Example:

```
GUITransform(694, 852, 844, 702,
             1, 1, 1, 1, 1 { Scaling              },
             0, 0          { Movement             },
             1, 0          { Visibility, Reserved },
             0, 0, 0       { Selectability        },
             \MeterParts\LinearLegend(Invalid, 0, 3, Invalid, 0, 0,
                                      0, 0, 0, 100));
```

ListAdd

Note: Deprecated. Do not use in new code.

(Alarm Manager module)

Description: This subroutine will add the alarm object to a list. This is useful if a user-defined list has been created.

Returns: Numeric

Usage: Script Only.

Function Groups: Alarm

Related to: Register (Alarm Manager) | CurrentTime

Format: \AlarmManager\ListAdd(AlarmObject, [EventTime], List);

Parameters:

AlarmObject

Required. The object value for the alarm that was passed to the Register subroutine.

EventTime

The time stamp to use when adding this event to the alarm lists. If invalid, the default is CurrentTime().

List

Required. Any numeric expression for the number of the list to which to add the alarm.

Comments: The ListAdd subroutine always returns "0".

Listbox

(System Library)

Description: Draws a list box with scroll bar (if required) and indicates the selected item.

Returns: Nothing

Usage: Steady State only.

Function Groups: Graphics

Related to: CheckBox | Droplist | GridList | HScrollbar | RadioButtons | Spinbox | SplitList | VScrollbar

Format: \System\Listbox(X1, Y1, X2, Y2, Data, Index, [Picked, Flat, DoubleClick, MaxLen, RightClick, PostIt, FocusID, Multi, PickList, ColWidths, ColLabels, ColDivider, EditSelected])

Parameters:

X1

Required. Any numeric expression giving the X coordinate on the screen of one side of the Listbox, usually the left side.

Y1

Required. Any numeric expression giving the Y coordinate on the screen of either the top or bottom of the Listbox, usually bottom.

X2

Required. Any numeric expression giving the X coordinate on the screen of the side of the Listbox opposite to X1, usually the right side.

Y2

Required. Any numeric expression giving the Y coordinate on the screen of the top or bottom of the Listbox, whichever is the opposite of Y1, usually the bottom.

Data

Required. An array of data to be displayed in the List-box.

Index

Required. The index of the highlighted item (i.e. any variable whose value will be set to the index of the chosen item (highlight).) If Multi is true, this will be the index of the last chosen item.

Picked

An optional parameter that may be a variable whose value is set to true (1) when an item is chosen in the Listbox. The setting of Index by an external source will not trigger Picked. If this information is not required and the next parameter is used, a value of invalid or a constant may be substituted.

Flat

An optional parameter that is any logical expression. If true (non-0) the border of the listbox will be a single black line, if false (0) it appear with a 3-D border. The default is false.

DoubleClick

An optional parameter that may be a variable whose value is set to true (1) when an item has been double clicked upon. If this information is not required and the next parameter is used, a value of invalid or a constant may be substituted.

MaxLen

An optional parameter that is any numeric expression giving the maximum length of the list. If omitted or invalid, the maximum list length is given by the size of the array Data.

RightClick

An optional parameter that may be a variable whose value is set to true (1) when an item is selected with the right mouse button.
If this information is not required and the next parameter is used, a value of invalid or a constant may be substituted.

PostIt

An optional parameter that is any logical expression. If true (non-0), a tool tip will be displayed if the text in the list box has been truncated at its right edge. The default is true.

FocusID

An optional parameter that is any numeric expression for the focus number of this graphic. If this value is zero, the list box will not accept keyboard input, but ouse input will still be recognized.
If it is less than zero, the Listbox will not accept any input and will appear grayed out. The default value is 1.
Values above 32767 are treated as if zero.

Multi

An optional parameter that is any logical expression. If true (non-0), multiple items may be selected in the list. The default is false.

PickList

An optional parameter that is a variable whose value is set to the list of items selected if Multi is true (1). If invalid, no items are selected.
This variable may initially be set to a dynamically allocated array (one created with the New function) containing items to be highlighted/selected upon the startup of the listbox.

ColWidths

An optional parameter that indicates the starting widths for multiple columns.

ColLabels

An optional parameter that provides an array of labels for the columns.

ColDivider

An optional parameter that indicates the type of divider to appear between columns. No border appears if the list is a single column. ColDivider may have one of the following values: The default is "2".

ColDivider	Divider Type
0	No column divider shown;
1	Show a non-moveable column divider. Or
2	Show a moveable column divider.

EditSelected

An optional flag that indicates whether a selected item may be edited in the list. EditSelected will be set to 0 when the editing stops. EditSelected may have one of the following values:

EditSelected	Meaning
Invalid	no editing;
0	enable item editing. Or
1	edit current selected item.

Comments:

This module is a member of the System Library, and must therefore be prefaced by \System\, as shown in the "Format" section. If you are developing a script application, use "System\..." rather than "\System\..." in the function call.

If Multi is true, multiple items in the list may be selected by using the <Shift> or <Ctrl> keys along with mouse input. If <Ctrl> is held while an item is clicked on by the mouse, it will become selected (or de-selected if it is already selected) and will be added to the list of chosen items. If <Shift> is held while an item is clicked on by the mouse, all items from the last selected item to the selected item will be selected. All other items outside of this list will be deselected. If both <Ctrl> and <Shift> are held while an item is clicked on by the mouse, all items from the last selected item to the selected item will be set to the state of the last selected item.

For any optional parameter that is to be set, all optional parameters preceding the desired one must be present, although they may be invalid.

Examples:

```
\System\Listbox(10, 10, 110, 210 { Outline of list box },
                List { Data to display },
                Highlight { Index of highlight });
```

In this example, the listbox is displayed with only a single black line around it. Notice that since none of the optional parameters are used, they have been omitted. Also, the listbox will display the full contents of the array because MaxLen has similarly been omitted.

```
\System\Listbox(20, 355, 295, 100 { Outline of listbox },
                NameList { list of names to display },
                ListIndex { Index of highlight },
                0 { Picked parm not used },
                0 { Drawn 3D },
                Double { TRUE when double clicked },
                Invalid { MaxLen parm not req'd },
                RtClick { TRUE when rt clicked on },
                1 { tool tips as req'd },
                5 { Focus ID },
                1 { Multiple selections },
                FinalList { Var containing list of
                selected items });
```

This example illustrates including all of the optional parameters, even those such as MaxLen that in this case are not used.

ListKeys

Description Returns an array of all keys used within a dictionary. It is expectd that this function will be used primarily in the context of metadata (extended information attached to a variable). ListKeys also enables you to discover what is in a dictionary.

Returns Array

Usage Script Only.

Function Groups Dictionary, Variable

Related to: Dictionary | MetaData | DictionaryCopy | DictionaryRemove | GetNextKey

Format: ListKeys(dictionary[, order, return value]);

Parameters

Dictionary

Required. The name of the dictionary.

Order

An optional numeric expression. Defines the search according to the following table of values. Defaults to 0 if missing or invalid

Order	Meaning
0	List in forward alphabetic order.
1	Ordered by when the keys were added to the dictionary with the oldest key first.
2	List in backward alphabetic order
3	Ordered by when the keys were added to the dictionary with the newest key first.
4	Sparse Numeric. Keys must be numeric. Used to return a sparse array in order based on the key values.

Return Value

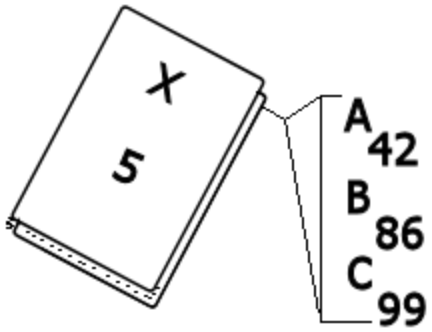
Controls what is returned according to the following

Value	Returns
< = 0	Invalid
1	1D array (vector) containing the dictionary keys
2	1D array (vector) containing the dictionary values, ordered by key
3	2D array (table). The first column (left-most) will contain the dictionary keys and the second (right-most) will contain the dictionary values. Ordered by key.
>= 4	Invalid

Example:

(given a dictionary named X as shown)

```
R = ListKeys( X );
R == ["A", "B", "C"];
```

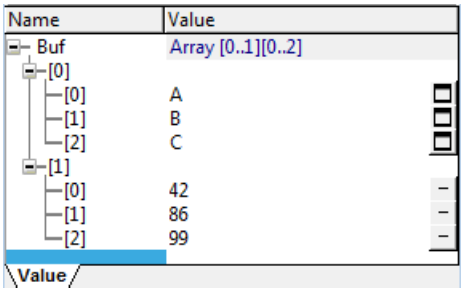


Example 2:

Given the code:

```
IF Watch(1);
[
  X = Dictionary(0, 5);
  X["A"] = 42;
  X["B"] = 86;
  X["C"] = 99;
  Buf = ListKeys( X, 1, 3);
]
```

Buff will contain:



ListRemove

Note: Deprecated. Do not use in new code.

(Alarm Manager module)

Description: This subroutine will remove the alarm object from a list. This is useful if a user-defined list has been added.

Returns: Numeric

Usage: Script Only.

Function Groups: Alarm

Related to: Register (Alarm Manager) | CurrentTime

Format: \AlarmManager\ListRemove(AlarmObject, [EventTime], List);

Parameters:

AlarmObject

Required. The object value for the alarm that was passed to the Register subroutine.

EventTime

The time stamp to use when adding this event to the alarm lists. If invalid, the default is CurrentTime().

List

Required. Any numeric expression for the number of the list from which to remove the alarm.

Comments: The ListRemove subroutine always returns a "0".

ListVars

Description: Returns a list of variables.

Returns: Array

Usage: Script Only.

Function Groups: Compilation and On-Line Modification, Variable

Related to: FindVariable

Format: ListVars(Module, Name, LowClass, HighClass, Type, Attributes, Global, Info, Sort)

Parameters:

Module

Required. Any module or object value. This identifies the module where the variable list begins.

Name

Required. Any text value, which specifies a name to match when listing. Wildcards are allowed. The wildcard "?" will match any character in a name. The wildcard "*" will match any series of characters.
The backslash character has special meaning, in that it can be added in front of the characters *, ? or \ in order to search for these characters explicitly. In these three cases, the leading backslash will be ignored.
Other characters must match exactly. A leading backslash in front of any characters other than the three noted will be included in the search.

LowClass

Required. Any numeric expression. This specifies the lowest class included in the list. Valid range is 0 to 65535. Default class for variables is 0. See also: Variable Classes.

HighClass

Required. Any numeric expression. This specifies the highest class included in the list. Valid range is 0 to 65535. Default class for variables is 0.

Type

Required. Any numeric expression that specifies the type of variable to match, as found by adding together values in the following table:

Type	Bit No.	Variable Type
0	-	Match all
1	0	Normal
2	1	Array
4	2	Parameter
8	3	Module

Attributes

Required. Any numeric expression that specifies the attributes of a variable to match, as found by adding together values in the following table:

Attributes	Bit No.	Variable Attribute
0	-	Match all
1	0	Shared
2	1	Persistent
4	2	Constant
8	3	Simple
16	4	Temporary
32	5	Protected

Global

Required. Any logical expression. If true, all variables in scope are listed (including parent modules). If false, only local variables in Module are listed.

Info

Required. Any numeric expression, which specifies the information to return in the array, as shown in the following table:

Info	Information
0	Text name
1	Attributes (bit field - same as for VarAttributes function)
2	Default value
3	Class
4	Module value where defined
5	Module text name where defined
6	Number of instances
7	Current value
8	Pointers to variables

If Info is 7 or 8, then Module should be the object value of the instance where the variable instances reside to get all information.
If Module isn't an object value, then the return value will be invalid unless there are shared or persistent variables.

Sort

Required. Any logical expression. If true, the variables are sorted in alphabetical order. If false, the variables are listed in the order in which they appear in the document file.

Comments: The variable used to store the returned array does not need to be declared as an array. It will be dynamically allocated as such by the function. If no variables are found, the return value will be a valid pointer to an array with no elements.

Data Handling in Statically-declared Arrays vs. Dynamically-allocated Arrays.

ListVars handles data in statically declared arrays differently from data stored in dynamically allocated arrays. In the following example, two arrays have been declared. The first (A1) is statically-declared, while the second (A2) is dynamically-allocated.

```
A1[2][3];
A2;

A2 = New(2, 3);
A1[0][0] = "zerozero";
A2[0][0] = "zerozero";
A1[1][0] = BuffStream("onezero");
A2[1][0] = BuffStream("onezero");
A1[1][1] = 11;
A2[1][1] = 11;
A1[1][2] = 12;
A2[1][2] = 12;

AFiles = ListVars(Self(), "A*", 0, 65535 { class
limits }, 0, 0 { all matches },
0 { local only }, 8 { ptr to variables }, 1 {
sort });
```

To retrieve the value of each element in the statically-declared array named, "A1", you can use:

```
ElemA1_00 = AFiles[0][0][0];
ElemA1_10 = AFiles[0][1][0];
ElemA1_11 = AFiles[0][1][1];
```

… and so forth. However, this same call for the dynamically-allocated array named, "A2", will not retrieve the expected elements. Rather, the correct syntax for data retrieval in this case is:

```
ElemA2_00 = (*AFiles[0])[0][0];
ElemA2_10 = (*AFiles[0])[1][0];
ElemA2_11 = (*AFiles[0])[1][1];
```

The data retrieval method displayed above for the dynamically-allocated array will also work for static arrays, thus it is recommended that this be the method used for general array data retrieval.

Example:

```
If ! Valid(myList);
[
  myList = ListVars(FindVariable("Calculations", Self(), 0, 1)
  { Module to find variables of },
  "*" { List all variables },
  0, 0 { Class 0 variables only },
  0, 0 { All types and attributes },
  0 { Variables of this module only },
  0 { Name of variable },
  1 { List alphabetically });
]
Table(myList[0] { Starting array element },
      ArraySize(myList, 0) { Number of elements },
      10, 10, 0, 10 { Start at (10, 10) list vertically },
      4, 0, 100, 14, 0, 0, 8, 0 { Text values and attributes });
```

This set of statements will display a list of all variables that belong to the module Calculations in alphabetical order in the upper left had corner of the window. Note that the first statement has the potential to become an "If 1" condition (infinite loop) if there are no variables belonging to module Calculations (i.e. the ListVars function returns Invalid).
In the next example, the ListVars statement is returning an array of pointers to all variables beginning with the letter A, which in this case is two arrays, one statically declared (A1) and the other dynamically allocated (A2):

```
A1[2][3];
A2;
…
A2 = New(2, 3);
A1[0][0] = "zerozero";
A2[0][0] = "zerozero";
A1[1][0] = BuffStream("onezero");
A2[1][0] = BuffStream("onezero");
A1[1][1] = 11;
A2[1][1] = 11;
A1[1][2] = 12;
A2[1][2] = 12;
AFiles = ListVars(Self(), "A*", 0, 65535 { class limits },
                  0, 0 { all matches }, 0 { local vars },
                  8 { ptr to variables }, 1 { sort });
```

The array AFiles will now contain a list of pointers to the two arrays. To access the individual elements of array A1 and A2, each element of array AFiles must be de-referenced using the * character:

```
ElemA1_00 = (*AFiles[0])[0][0];
ElemA1_10 = (*AFiles[0])[1][0];
ElemA2_00 = (*AFiles[1])[0][0];
```

It is interesting to note that while this method may be used on either array, the following statement format will only set the variables to the expected value in the case of the statically declared array (A1):

```
ElemA1_00 = AFiles[0][0][0];
```

Ln

Description: Returns the natural logarithm (base e) of a value.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Generic Math

Related to: Exp | Log | Pow

Format: Ln(X)

Parameters:

X

Required. Any numeric expression. The value must be strictly greater than 0 for the result to be valid.

Comments: This is the inverse function to Exp.

Example:

```
a = Exp(23);
b = Ln(a);
```

In this example, b will be equal to 23.

LoadDLL

Description: Loads a Microsoft Windows™ dynamic link library.

Note: 64-bit VTScada can load only 64-bit DLLs.
32-bit VTScada can load only 32-bit DLLs.
VTScada Internet Clients can load only 32-bit DLLs

regardless of whether the VTScada server is 32-bit or 64-bit.

Returns: DLL handle, numeric error code, or Invalid. See comments

Usage: Steady State only.

Function Groups: Compilation and On-Line Modifications, DLL, Software and Hardware

Related to: DLL

Format: LoadDLL(FileName[, VICRemoted])

Parameters:

FileName

Required. Any text expression for the file name of the DLL to load. If the path is not specified the standard WindowsÔ search pattern for DLLs will be used.

VICRemoted

(Optional). A parameter that indicates whether or not the handle returned by LoadDLL will be remoted to the VIC.

If VICRemoted is a valid, non-zero, positive integer, and the LoadDLL function is being executed in a module instance that is within a VIC session, then all DLL calls using the handle that LoadDLL returns will be remoted to the VIC.

Whether or not the LoadDLL is in a VIC session is determined by the call tree from the module instance running the LoadDLL call. The root of the call tree must be an instance of BrowserClient. In practical terms, this means that you cannot expect session-aware DLL calls to function if you explicitly launch a module with a caller other than a module instance within the VIC session tree.

Defaults to 0 if not otherwise specified.

Comments:

The return value is either the Windows™ handle of the new DLL (ValueType == 38), or on failure, the least significant 16-bits of operating system error code (ValueType < 3). There is no dynamic download of a DLL to the VIC. DLLs must be manually installed on the VIC in either the system program folder (e.g. \Windows\System 32), or in a folder that is on the executable search path (the PATH environment variable).
LoadDLL is a steady-state statement. If it stops, all VALUEs holding a copy of the DLLHandle returned from the LoadDLL statement will be invalidated, and if there are no other LoadDLLs running for the same DLL, the DLL will be unloaded. (LoadDLL is not an expensive operation, but it is good programming not to do this very often.)

Note: LoadDLL will block the calling interpreter thread until the DLL loaded has completed on the VIC. This means that nothing else will execute on the VIC session thread until the DLL load is complete. This has the advantage (from the programmer's perspective) that the DLL handle returned is immediately ready for use. LoadDLL will return Invalid if there is a problem with the parameters, an integer error code if it cannot load the DLL, or a DLLHandle (ValueType ==38) if the load was successful. On failure, the return value will be the least significant 16-bits of operating system error code (ValueType < 3).

DLL calls do not need to consume the return value, or have no return value should be written as such (i.e. explicitly specify a return type of zero to indicate that there is no return value, or that the return value is irrelevant. If you specify other than zero for the return type, the DLL call will block the calling interpret thread until the VIC has run the DLL call and returned the value. This means that scripts

that rely on synchronous DLL calls will still work, even with the DLL remoted, however, they will slow down. If you specify a return value type of zero, a remoted DLL call is dispatched to the VIC asynchronously (i.e. your interpreter thread will not block). If, for example, you have a sequence of DLL calls that will execute asynchronously, followed by one that is synchronous (because you want the return value), the calling interpreter thread will block only on the synchronous call. This may take a little longer because the order of the DLL calls is preserved on the VIC, and all the asynchronous ones will have to complete execution before the synchronous one can be executed. This is much quicker than using all synchronous calls, because the asynchronous ones are batched together and require no reply (other than a batch acknowledgment from the VIC), reducing comms latency.
DLL handles should not be declared as shared if they are going to be remoted. Shared DLLs will only be created once, and so will only operate on the server OR client, depending upon which one started them first.

LoadMIB

Description: Loads a specified MIB or set of MIBs and returns a dictionary describing the hierarchy of the MIBs.

Returns: Dictionary

Usage: Script

Function Groups: File I/O, Software and Hardware

Related to:

Format: LoadMIB(MIBPath[, SubIDIndexing, LoadDescriptions, ErrorOut])

Parameters:

MIBPath

Required. Apath to any single MIB or a directory containing MIBs (possibly in further subdirectories). The returned MIB dictionary will also include those elements of the base MIBs found in <VTSInstallDir>\MIBS that are referenced in the MIB found in MIBPath. If Invalid, then the returned MIB dictionary will only include the base MIBS.

SubIDIndexing

An optional Boolean expression. If TRUE, then items in the returned MIB dictionary are keyed by numeric portion of the OID for that element. If FALSE, then the items are keyed by the label for the element. Defaults to TRUE.

LoadDescriptions

An optional Boolean expression. If TRUE, then the returned MIB dictionary will contain any description for the element (which may be lengthy). If FALSE, then no description is loaded. The default is TRUE.

ErrorOut

An optional variable into which any errors found in parsing the MIBs will be returned. Will take the form of a linked list of error message structures. The message structure will contain two fields: Error, containing the error text and Next containing the next error message structure, if any.

Com-
ments:

Label	The textual "name" of the element.
SubID	The portion of the OID corresponding to the element.
OID	The full OID for the element.
Desc	The textual description of the element (if the LoadDescriptions parameter is TRUE).
Syntax	The type of the element, e.g. "OCTET STRING" or "INTEGER" etc.
Type	The numeric type of the element, e.g. INTEGER is 3. The full list of possible types is: OTHER0 OBJID1 OCTETSTR2 INTEGER3 NETADDR4 IPADDR5 COUNTER6 GAUGE7 TIMETICKS8 OPAQUE9 NULL10 COUNTER6411 BITSTRING12 NSAPADDRESS 13 UINTEGER14 UNSIGNED3215 INTEGER3216 SIMPLE_LAST16 TRAPTYPE20 NOTIFTYPE21 OBJGROUP22 NOTIFGROUP23 MODID24

LoadModule

Description: Loads a module from its .RUN files and returns a pointer to that module.

Returns: Module

Usage: Script Only.

Function Groups: Compilation and On-Line Modifications, Advanced Module

Related to: Launch | Thread

Format: LoadModule(FileName, Library, [ModuleName, LoadNow, AppGUID, RefModule, TestCodeLoad])

Parameters:

FileName

Required. Any text expression giving the name of the .RUN file from which to load the module.

Library

Required. Any module value that indicates the library module to which the loaded module will belong.

ModuleName

An optional parameter that is the text expression giving the module's name. This name will then be associated with the module when it is displayed in certain situations, such as in the debugger and the module tree diagram. If this parameter is invalid, the module will appear as "System" when its name is displayed.

LoadNow

An optional parameter that is any logical expression. If true (non-0), the module and its entire sub-tree are loaded immediately. If false (0), the module and its sub-tree are loaded on demand. The default is false.

AppGUID

An optional parameter that should be included and hold the text GUID of the application when the LoadModule is loading an application.

RefModule

An optional parameter that has been reserved for use by the Test Framework and is used when a module is being loaded in isolation of the application module tree.
Its presence allows source file offsets returned by the TextOffset statement to operate off the loaded module.

TestCodeLoad

An optional parameter that has been reserved for use by the Test Framework to indicate that the LoadModule is loading actual code to be run.
It can contain any Boolean expression that evaluates to TRUE to indicate that the code to run is being loaded.

Comments: The return value is a pointer to the module. It can be used in a Launch or Thread statement.
The usage of the AppGUID and RefModule parameters determines how the VTScada engine will treat the module load:

- If AppGUID and RefModule are both Invalid, this is a "normal" load. No special processing is performed.
- If AppGUID is a text GUID and RefModule is absent or Invalid, the load is a load of the root module for the application. This causes the loading module to become the "namespace root" for the application.
- If RefModule is valid, the loading module will cross-reference the reference module, enabling the TextOffset statement to return the correct source file offsets for the loading module. The RefModule must be part of the complete module tree for the application. It is also used to ensure that breakpoints are re-

made when the test module reloads.

Example:

```
If ! Valid(CompiledModule);
[
   ExpressionTemplateRUN = RunFileName( FindVariable( "Expres-
sionTemplate", Self, 0, 0));
   CompiledModule = LoadModule( ExpressionTemplateRUN,  Expres-
sionParentModule, "Compiled Expression");
]
```

LocalGroup

Description Returns an indication of whether the current Windows™ user is a member of the specified local group. LocalGroup interrogates only local groups, not domain groups.

Returns Numeric

Usage Script or steady state.

Function Groups Security

Related to:

Format: LocalGroup(Group)

Parameters

Group

Required. The group parameter, which is a value from 0 to 5. These values are defined in the following table:

Group	Local Group
0	Administrators
1	Backup Operators
2	Guests
3	Power Users
4	Replicator
5	Users

Comments This function returns an indication of whether the current Windows™ user is a member of the specified local group. Its return values are outlined in the following table:

Return Value	Information
0	Not a member
1	Is a member
2	Not applicable

Invalid Parameter or internal error:
The user must log off and log on if a local group was changed for the change to be noticed by the LocalGroup function call. (This behavior is inherent to Windows and is not due to a VTScada feature or defect.)
Custom local groups (local groups that are not of type Administrators, Backup Operators, Guests, Power Users, Replicator or Users) cannot be interrogated by this function.

LocalScope

Description: Equivalent to Scope(Obj, Name, TRUE).
LocalScope is a useful shortcut where the second parameter is not a constant string.

Returns: Reference

Usage: Script or steady state.

Function Groups: Basic Module

Related to: Variable | ScopeLocal | Scope

Format: LocalScope(Obj, Name)

Parameters:

Object

Required. Any expression for the object value where Member may be found.

Member

Required. Any text expression for the member name.

Comments: This function is the same as the '.' operator, when the '.' operator is used between two operands. (Object.Member).
As an example, the LocalScope() function is useful for referencing a tag object where its name contains special characters or spaces:

```
TagObj = LocalScope(\VTSDB, "R&R Level")
```

Example:

```
TagName = "MyTag";
TagObj = LocalScope(\VTSDB, TagName);
```

Returns a reference to the given tag object, found within the current module.
This is the equivalent to:

```
\VTSDB.MyTag
```

Locate

Description: Locates a text string, returning the offset of the first matching string in a buffer.

Returns: Numeric

Usage: Script or steady state.

Function Groups: String and Buffer

Related to: CharCount | BuffWrite

Format: Locate(Buffer, Offset, Search [, Method])

Parameters:

Buffer

Required. The text expression to search.

Offset

Required. Any numeric expression giving the buffer offset (in characters or bytes), from which to start searching (i.e. to start at the beginning of the buffer, set this parameter to 0).

Search

Required. Any text expression for which to search.

Method

An optional numeric expression that controls how the Search parameter is interpreted as per the following table. The default value for Method is "0".

Method	Description
0	Treat Search as the exact string for which to search (strstr).
1	Treat Search as the case-insensitive string for which to search (stristr).
2	Treat Search as a set of characters to match against (strpbrk).
3	Treat Search as a case-insensitive set of characters to match against.

Comments:

This function returns the buffer offset of the first string matching Search. If no match is found, or if the length of Search plus Offset is larger than the length of Buffer, -1 is returned. The return value could be used in the Offset parameter to perform successive searches.

This function can be used to perform fast table searches. Build a table in a text variable using BuffWrite. Make sure that all entries in the table are the same length. You can now use Locate to find the buffer offset of a matching text string.

Locate() supports case-insensitive search, which is spe-

cified by using Offset < 0. The real start is then calculated by subtracting it from -1.

Examples:

```
w = Locate("abcABCabc", 0, "bc");
x = Locate("abcABCabc", 2, "bc");
y = Locate("abcABCabc", 0, "Bc");
z = Locate("abcABCabc", 0, "X");
```

The values of w, x, y and z will be 1, 7, -1 and -1 respectively.

LocCapture

Description: Capture Locator Input. This statement captures all subsequent input from the locator device and routes it to a specific window.

Returns: TRUE if capturing. Invalid otherwise.

Usage: Steady State only.

Function Groups: Locator, Window

Related to: WinXLoc | WinYLoc | XLoc | YLoc

Format: LocCapture(Object, Enable)

Parameters:

Object

Required. Any expression that returns an object value of a window or an object within a window whose coordinate system is to be used for locator coordinate reports until capture is released.

Enable

Required. Any logical expression. If true (non-0), the locator input will be captured by the window containing the module expressed by Object. If false (0), the locator capture is yielded.

Comments: This statement captures all locator input and routes it to

the window containing the module instance that issued the LocCapture statement. This is most useful when there are child windows present that are graphically in front of the window with capture. With capture disabled (the normal condition), the coordinates returned to a WinXLoc(Self( )) would change from being relative to the parent window's coordinate origin to the child window's coordinate origin as the locator moved over the child window. With capture enabled, the coordinates would always be relative to the parent window's coordinate origin, no matter which win-dow the locator was over.

Example:

```
LocCapture(Self(), 1);
```

LocSwitch

Description: Returns the current status of the locator (mouse) buttons over the window which contains the LocSwitch statement.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Locator

Related to: Click | SetXLoc | SetYLoc | Target | WinLocSwitch | XLoc | YLoc

Format: LocSwitch()

Parameters: None

Com-ments: The function has no parameters and therefore the empty parentheses following the function name are optional.

If the locator is not installed, or if it is over a window other than the one containing the module with the LocSwitch statement in it, the function returns 0.

Unlike the WinLocSwitch statement, this function is not triggered by mouse clicks in parent or child windows, only those occurring over the owning window. The return value has the following significance:

Return Value	Mouse Button(s)	No. of Clicks
0	No buttons	-
1	Right button	Single
2	Middle button	Single
3	Right & middle button	Single
4	Left button	Single
5	Left & right button	Single
6	Left & middle	Single
7	All three buttons	Single
8	No buttons	-
9	Right button	Double
10	Middle button	Double
11	Right & middle button	Double
12	Left button	Double
13	Left & right button	Double
14	Left & middle	Double
15	All three buttons	Double

Example:

```
If LocSwitch() == 4 NextState;
```

This statement causes a state change to NextState at the press of the left mouse button.

Log

Description: Returns the common logarithm (base 10) of a number.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Generic Math

Related to: Exp | Ln | Pow

Format: Log(X)

Parameters:

X

Required. Any numeric expression. The value must be strictly greater than 0 for the result to be valid.

Comments: The common antilogarithm can be found using the Pow function.

Example:

```
n = Log(10000);
```

The variable n will be set to 4.

LogNTEvent

Description Logs events to the system event log.

Returns Nothing

Usage Script or steady state.

Function Groups Log

Related to:

Format: LogNTEvent(Severity, Strings [, Source, Category, EventID, DataSize, Data, UNCServerName])

Parameters

Severity

Required. A numeric code indicating the type of event to log. Severity can be one of:

Severity	Event
0	Informational
1	Warning
2	Error
3	Audit Success (Security Event)
4	Audit Failure (Security Event)

String

Required. A single string or an array of strings to pass to the event as parameters.

Source

An optional parameter indicating the name of the source program. The default value for Source is "VTScada" (see comments section).

Category

An optional parameter indicating the numeric ID of event category. The default value for Category is "0", indicating "none" (see comments section).

EventID

An optional parameter indicating the numeric event code. The default value for EventID is "1001" (see comments section).

DataSize

An optional parameter indicating the size of binary data in bytes (see comments section).

Data

An optional parameter indicating the binary data to

store with the event (see comments).

UNCServerName

An optional string specifying the UNC server name for Source (see comments).

Comments

Event logs store important events for applications running on Windows. Because the logging function is designed to be general purpose, you must decide what information is appropriate to log. As a general rule, you should only log information that could be useful in diagnosing a hardware or software problem. The event logging facility is not intended to be used as a tracing tool.

Event logging consumes system resources such as disk space and processor time. The amount of disk space that an event log requires depends on how much information you choose to log. For this reason, it is important to log only essential information.

Following, are some Microsoft guidelines regarding the types of events you may wish to log for each severity

Informational Information events indicate significant successful operations that occur infrequently. It is not generally considered appropriate for an application to log an event each time it starts.

Warning Warning events indicate problems that are not immediately significant, but that may indicate conditions that could cause problems in the future. Generally, if an application can continue or recover from an event without loss of functionality or data, it can classify the event as a warning.

Error Error events indicate significant problems about which the user should know. Error events usually result in or from the loss of functionality or data.

Audit Failure When a security access attempt fails, it is considered an audit failure. A failed logon attempt is a failure audit event.

Audit Success When a security access attempt succeeds, then it is a success audit event. For example, a successful logon attempt is a success audit event.

For more information on NT Event Logging, please see the Microsoft MSDN documentation.

The only required parameters for LogNTEvent are the severity code (Severity) and the string or array of strings to log with the event (Strings)

You may optionally specify:

- The application event Source (the default value is "VTScada").
- The Category (numeric starting at "1", where "0" is the "none" category). The default is "0" (none).
- The EventID (numeric, generally starting at 1000 or so in order to not conflict with the category numbers. The first and only defined EventID for source "VTScada" is "1001", which simply displays the passed-in string).
- Data and Datasize (used to store binary data with the event). DataSize should be the size, in bytes, of the Data array. Data is any array of binary data to be stored along with the event. And
- UNCServerName, which is the machine to which to log the event (the default is Invalid, which results in the event being logged on the local machine. Otherwise, you may specify the UNC name for the machine to which you would like the event logged).

Note that specifying values other than the defaults for Source or EventID will result in the event log displaying the event improperly, unless a custom DLL is written to handle the case. However, this does not prevent the event from being logged, and a custom DLL can be added at a later time.

Example:

```
databuff = MakeBuff(10, 65);
textstring = "RTU 0015 is offline";
res = LogNTEvent(1,
                 textstrings,
                 Invalid {source},
                 Invalid {category},
                 Invalid {EventID},
                 10,
                 databuff);
```

This example will log a warning event under the source name "VTScada", with default category ("none"), and default event ID (1001). The details of the event will be the string "RTU 15 is offline, and attached to the event is 10 bytes of binary data – an array of 10 letter A's.

LogOff

Security Manager Module

Description	Logs the calling user session off.
Returns	Nothing
Usage	Script Only.
Related to:	AlternateIdCheck | AlternateLogoff | AlternateLogon | Authenticate | QuietLogon | UserCredChange | UserLogonDialog
Format:	\SecurityManager\LogOff()
Parameters	None
Comments	After logging off, the user session reverts to the Logged Off

user.

LookUp

Description: Looks up a value in an array and returns the index of the element containing that value.

> Note: This function replaces the deprecated TextSearchList and SearchForListItem system modules.

Returns: Numeric

Usage: Script or Steady State

Function Groups: Array

Related to: AMax | AMin | ArrayDimensions | ArraySize | ArrayStart | AValid | Sort | Sum | TextSearch

Format: Lookup(ArrayElem, N, Match [, CaseInsensitive])

Parameters:

ArrayElem

Required. Any array element giving the starting index for the array operation. The index for the array may be any numeric expression.
If processing a multidimensional array, the usual rules apply to decide which dimension should be examined

N

Required. Any numeric expression giving the number of array elements to compute. If N extends past the upper bound of the lowest array dimension, this computation will "wrap-around" and resume at element 0, until N elements have been processed.

Match

Required. Any value of any type to look for in the array. In the case of text, the search is case sensitive. If

this value is invalid, the return value will also be invalid.

CaseInsensitive

An optional parameter that is any logical expression. If true (non-0) and the array contains text strings, the comparison will not be case sensitive. If false (0), the comparison will be case sensitive for text strings. The default is false.

Comments: It is acceptable for the array to contain invalid values - these will be skipped over in the search for Match.
The return value will be invalid if the value of Match isn't found in the array.
Note that TextSearch should be used for large arrays due to its considerably faster search algorithm.

Example:

```
index = Lookup(labels[0], ArraySize(data, 0), "off", 1);
```

This searches the array labels for a string of any case that matches the string "off". If a match is found, its value is assigned to index.

LValue

Description: Left-hand Side Value. This function returns an indication of whether its argument can be used on the left-hand side of an assignment.

Returns: Boolean

Usage: Script or steady state.

Function Groups: Compilation and On-Line Modifications, Generic Math, Variable

Related to:

Format: LValue(Expr)

Parameters:

Expr

Required. Any expression.

Comments: This is a compiler function that returns 1 (true) if data can be stored into Expr (i.e. if it is a variable, etc.) and 0 otherwise.

M Functions

The sections that follow identify all VTScada functions beginning with "M".

MACID

Description: Enumerates and returns the MAC IDs registered on a particular machine.

Returns: A 2D array containing network card names and MACIDs

Usage: Script only.
May be used in optimized Tag Parameter Expressions.

Function Groups: Software and Hardware

Related to:

Format: MACID()

Parameters: None

Comments: MACIDs are unique to the NIC cards present within the machine but are NOT guaranteed to be unique for the virtual NICs generated for things like virtual machines or NIC emulations.
The name of the resource that generated the ID is provided

with each ID in order to help sift out faux MACs. Note that Windows appears to always provide the real MACs first, and since almost all machines have a built-in NIC the first MAC can be assumed to be real.

Example:

```
MACIDInfo =  MACID();
```

On a machine, which has two network cards, MACIDInfo will have (for example):

```
MACIDInfo[0][0] == Realtek RTL8169/8110 Family PCI Gigabit Ethernet
NIC (NDIS 6.0)
MACIDInfo[0][1] == Realtek PCIe GBE Family Controller
MACIDInfo[1][0] == a buffer containing the binary MAC address for the
card named in MACIDInfo[0][0]
MACIDInfo[1][1] == a buffer containing the binary MAC address for the
card named in MACIDInfo[0][1]
```

MakeBitmap

Description: Loads an image file of types BMP, EMF, WMF, APM, CUT, PCX, JPG, PNG, or TIF into memory and returns a handle to the result. Returns Invalid upon failure.

Returns: Image handle

Usage: Script or steady state.

Function Groups: Graphics

Related to: BitmapInfo | Crop | GUIBitmap | GUIButton | ImageArray | ImageSweep | ModifyBitmap

Format: MakeBitmap(FileName [, Transparent1, Transparent2])

Parameters:

FileName

Required. Any text expression giving the name of the file containing the image. A known path Known Path Aliases for File-Related Functions may be provided in the form, :{KnownPathAlias}.

Transparent1

An optional parameter giving the first color value to make transparent. Any of the following may be used:

- a palette index VTScada Color Palette
- a system color (constant)
- an RGB string in the format, "<RRGGBB>"

If this value is negative 1 (-1), this parameter is ignored and no first color is made transparent.

Transparent2

An optional parameter giving the second color value to make transparent. Any of the following may be used:

- a palette index color
- a system color (constant)
- an RGB string in the format, "<RRGGBB>"

If this value is negative 1 (-1), this parameter is ignored and no second color is made transparent.

Comments: This function creates and returns an image value by loading an image from a file. All images are rendered in 32-bit color. An indexed color must translate into an exact 32-bit match in order to cause transparency. Colors may have their own transparency data via alpha values. Using an image value can improve speed and reduce memory requirements over using a file name directly in a function or statement.

Example:

```
Image = MakeBitmap("C:\Bitmaps\dial.bmp");
```

This will create an image value storing the image from the file "C:\Bitmaps\dial.bmp".

MakeBuff

Description: Creates a buffer and returns its address.

Returns: Address

Usage: Script or steady state.

Function Groups: String and Buffer

Related to: BuffOrder | BuffRead | BuffStream | BuffToArray | BuffToParm | BuffToPointer | BuffWrite | MakeFixedBuff

Format: MakeBuff(Length, Value)

Parameters:

Length

Required. Any numeric expression giving the length (number of bytes) of the buffer to create. This value must be between 0 and 0x7FFFFFFF.

Value

Required. Any numeric expression giving the initial value from the ASCII Character Set for every byte in the new buffer (see "ASCII Character Set"). It must be in the range 0 to 255.

Comments: This function can be used to create buffers for BuffWrite or similar functions that require an existing buffer. The return value is an address to a buffer of Length bytes, with each byte equal to Value. If the two parameters are integer constants and the buffer length is no more than 256 bytes, the compiler will convert this statement internally to a constant text string for speed at execution time.

The maximum buffer size is 0x7FFFFFFF (2,147,483,648) characters. Any size larger than this will result in an invalid value.

Example:

```
buff1 = MakeBuff(5, 0x41);
```

This will cause buff1 to be assigned the address where the value "AAAAA" is stored.

MakeCall

Modem Manager

Description This subroutine queues a call request.

Returns Numeric

Related to:

Format: \ModemManager\MakeCall(PhoneNumber, Baud, DataBits, StopBits, Parity [, CallTime, Tag, Workstation, MediaMode, Voice, UseLocal, Area, UserData, Service, Attempts, QTime, ID, InitString, UserName, Password, Domain, PPPFlags]);

Usage Script Only.

Parameters

PhoneNumber

Any text expression for the phone number to be called. If required, this may be the canonical international format.

Baud

Any numeric expression giving the baud rate to be used on the modem connection. The baud rate must be in the range of 0 to 115200, and must divide evenly into 115200 with no more than a 2.5% error. The value of "0" has special significance. This corresponds to the maximum baud rate available for the particular device. See Baud Rate.

DataBits

Any numeric expression giving the number of data bits per character to be used on the modem connection. DataBits must be 5, 6, 7, or 8.

StopBits

Any numeric expression giving the number of stop bits per character to be used on the modem connection. StopBits must be 1 or 2.

Parity

Any numeric expression giving the parity checking to be used on the modem connection. This may be one of:

Value	Parity
0	No parity
1	Odd parity
2	Even parity
3	0 Stick (space parity)
4	1 Stick (mark parity)

CallTime

A signed, numeric expression representing the time in seconds, relative to the current time, at which the call should be made. A negative value means that the call should be started as soon as possible; however, the call will be queued behind calls that specified a more negative time. If the value CurrentTime() is used here, then the call will go to the head of the queue. The default value for this parameter is "0" (i.e. now).

Tag

Any expression resolvable to the name of the tag making the call. The default is Caller(Self())\Name. Tag must be resolvable in the scope of \Code, and must resolve to an object. That object must contain a variable called DataPort that will be used by the Modem Manager for call control and progress purposes.

Workstation

Any text expression identifying the computer on whose behalf the call is to be made. The call progress and results will be delivered to the DataPort variable in the object \Code\Tag running in the application identified by the GUID of the application that originally called MakeCall() on the workstation identified by the Workstation parameter (but see also the Service parameter). This parameter defaults to the value \RPCManager\WkstnName on the machine that calls MakeCall ().

MediaMode

A valid numeric expression being a valid MediaMode constant. Use this when you require particular media properties (e.g. Voice) of a modem, and not all modems in the pool posses those properties. The default value selects any available modem.

Voice

If making an outgoing voice call, then this parameter specifies the text GUID of the voice to be used by the Text-to-Speech engine.

UseLocal

Normally, calls are queued for dispatch via one of the pool modems, which may be on a completely different machine. If you wish to use a local modem (e.g. a USB or PC Card modem on a laptop computer), then this parameter should be set to a numeric, non-zero value. It is necessary to also configure at least one modem as a local modem.

Area

The area parameter of modem tags may be used to create functional groupings. If this parameter is specified to MakeCall(), then only a modem that has the same area parameter will be used to make the call. If there is no such modem configured in the system, then the

call will be unceremoniously cancelled.

UserData

Opaque user-supplied data that is passed back to the user when the ModemControl plug-in is called.

Service

The parameter Workstation explains the normal rules for delivery of the call results. If this Service parameter is given, and is the text name of a driver service, then the call results will be delivered to the current server for that service, rather than the machine identified by the Workstation parameter.

Attempts

Count of attempts at this connection. This parameter should be INVALID or 0 initially.

QTime

The time that the call was originally requested. This parameter should be INVALID initially. It is used on subsequent attempts.

ID

If valid, then this is the Server's call ID.

InitString

An optional text value that will be sent to the modem as the final part of initializing it for use, just before it dials out. The string should include any terminating characters that the modem will require to complete the string, such as "\r".

UserName

An optional text value providing the user name for remote authentication of a PPP connection.

Password

An optional text value providing the password for remote authentication of a PPP connection.

Domain

The domain name, used for remote authentication of a PPP connection.

PPPFlags

An optional numeric value indicating parameters of the PPP connection. See PPPDial for a list of values and their meanings.

Comments

Returns one if the call is successfully queued. If a zero is returned, the call has not been queued, and will not be queued.
As soon as MakeCall() is called, the tag's DataPort variable is set to a valid value. This value will be changed many times subsequently, but will remain valid until the call is completed or canceled.

Shortly after calling MakeCall(), DataPort will become a pointer to an array. This indicates that the call is queued. When call setup is initiated, DataPort will become an integer value >= "0". Should the call fail in any way, then DataPort will become negative.

If the call setup completes successfully, then DataPort will change to a Stream value (ValueType(DataPort)==8). The call requestor may now read and write to that stream to communicate with the called party. To hang-up the call, call CloseStream(). If the other end hangs up or the call fails, then DataPort will go invalid.

If the call setup fails, then the call will be retried according to the configured retry settings. If the call is retried, then DataPort will once more become an array pointer while the call is queued. DataPort will not go Invalid until the call is abandoned.
If the originator requires that a call be canceled before it is

completed, then the CancelCall() method should be used. If the originator changes the DataPort value, the effect is undefined.

Related Information:

. Refer to Call Progress and Error Codes. in the VTScada Programmer's Guide.

MakeDAG

Description: Constructs a Directed Acyclic Graph (DAG - an internal function representation).

Warning: For use by advanced programmers only. Irrevocable alteration of your application may occur.

Returns: Nothing

Usage: Script Only.

Function Groups: Compilation and On-Line Modifications

Related to:

Format: MakeDag(Opcode, Parms)

Parameters:

Opcode

Required. Any numeric expression for the opcode of the DAG to construct.

Parms

Required. Any numeric expression that indicates the number of additional parameters to be included in the new DAG. This is normally 0, except in cases such as a new FWrite DAG.

Comments: This function constructs a Directed Acyclic Graph (an internal function representation). This may be used to build up a function or statement without compiling. This is only intended to be used to define a new VTScada code syn-

tax.

MakeEditor

Description: Returns an editor value which is used by an editor

Returns: Editor

Usage: Script Only.

Function Groups: Editor

Related to: AddEditorText | CurrentLine | Editor | ForceEvent | GoToOffset | SetEditMode

Format: MakeEditor()

Parameters: None

Comments: This function is used by all of the functions that require an editor.

Example:

```
aNewEditor = MakeEditor();
AddEditorText(aNewEditor, "The start of my editor");
```

These functions will create an editor called aNewEditor and put the above text string in it.

MakeFixedBuff

Description: Creates a buffer value which has its data stored at a specific memory address. Not supported under 64-bit VTScada.

Returns: Nothing (uses first parameter)

Usage: Script or steady state.

Function Groups: Memory I/O, String and Buffer

Related to: BuffOrder | BuffRead | BuffStream | BuffToArray | BuffToParm | BuffToPointer | BuffWrite | MakeBuff

Format: MakeFixedBuff(ReturnBuff, Address, Size)

Parameters:

ReturnBuff

Required. Any variable in which the newly created buffer is stored.

Address

Required. Any expression which gives a 4 byte selector:offset value for the protected mode memory to access. This is usually returned from a DLL call and is handled as a long integer type value.

Size

Required. Any numeric expression giving the number of bytes in the buffer. This value must be between 0 and 65 500.

Comments: This statement is used to access specific addresses in protected mode memory. This is intended to be used for specialized inter-program communication tasks such as accessing a shared memory buffer used by another program.

MakeNonPersistent

Description: Takes a variable and makes it not persistent.

Returns: Nothing

Usage: Script or steady state.

Function Groups: Compilation and On-Line Modifications, Variable

Related to: AddVariable | ChangePersistentSize | FindVariable | MakeNonShared | MakePersistent | MakeShared | PersistentSize

Format: MakeNonPersistent(Variable)

Parameters:

Variable

Required. Any expression for the variable value. This value is typically returned from a FindVariable or an AddVariable call.

Comments: A persistent variable saves its current value on disk and is automatically restored upon restarting the module. This function will also make the variable not shared. See MakeNonShared for the results of making the variable a non-shared value. The variable will be removed from the persistent (.VAL) file as well.

MakeNonShared

Description: Takes a shared variable and makes it not shared.

Returns: Nothing

Usage: Script or steady state.

Function Groups: Compilation and On-Line Modifications, Variable

Related to: AddVariable | FindVariable | MakeNonPersistent | MakePersistent | MakeShared

Format: MakeNonShared(Variable)

Parameters:

Variable

Required. Any expression for the variable value. This value is typically returned from a FindVariable or an AddVariable call.

Comments: A shared variable has the same value for all instances of its owning module. An instance will be created for the variable in every running instance where the variable is used. The instance will be initialized with the value of the shared variable.
If Variable is a module value, it will be unaffected.

MakePersistent

Description: Takes a variable and makes it persistent (static).

Returns: Nothing

Usage: Script or steady state.

Function Groups: Compilation and On-Line Modifications, Variable

Related to: AddVariable | ChangePersistentSize | FindVariable | MakeNonPersistent | MakeNonShared | MakeShared | PersistentSize

Format: MakePersistent(Variable, Size)

Parameters:

Variable

Required. Any expression for the variable value. This value is typically returned from a FindVariable or an AddVariable call.

Size

Required. Any numeric expression giving the number of bytes of storage allocated in the .VAL persistent variable file for this variable.
For array types, set this to the byte size of the largest array element (normally 8 bytes for numeric values).
For arrays containing text, enter the character length of the longest string element.

Comments: Since a variable can not be persistent without being shared as well it will make Variable shared. (see MakeShared for the results of this). If the variable is already persistent,. there will be no change made to the variable. In particular, the persistent size of the variable will not change.

MakeShared

Description: Takes a variable and makes it shared.

Returns: Nothing

Usage: Script or steady state.

Function Groups: Compilation and On-Line Modifications, Variable

Related to: AddVariable | FindVariable | MakeNonPersistent | MakeNonShared | MakePersistent

Format: MakeShared(Variable)

Parameters:

Variable

Required. Any expression for the variable value. This value is typically returned from a FindVariable or an AddVariable call.

Comments: There will only be one instance of the variable's value so anywhere the variable is set it will set it for all uses. Care must be exercised that by making a variable shared you do not create a "double set" on the variable (i.e. setting it in two different statements at the same time). If you do the result will be that the variable will become invalid. For more information on double sets see the "Variable" section in Chapter 3.
If Variable is a module value, it will be unaffected.

MapDraw

Description: Draws a "slippy" map, showing a list of site tags as pins on that map.

Returns: Object reference

Usage: Steady State only. (called from a GUITransform)

Function Groups: Graphics

Related to: SlippyMapRemoteTileSource1 | GetSessionContainers

Format: MapDraw([Latitude, Longitude, Sites, InitZoom, MinZoom, MaxZoom])

Parameters:

Latitude

Optional numeric expression specifying the Y-axis center of the map. If not provided, the calculated center of all sites to be drawn on the map will be used.

Longitude

Optional numeric expression specifying the X-axis center of the map. If not provided, the calculated center of all sites to be drawn on the map will be used. If there are no sites to display on the map, the initial display will be centered on North America.

Sites

Optional array of sites to display. Each site item in the array must be a structure, containing valid pointers for latitude, longitude and a callback object. See comments.

InitZoom

Optional numeric expression, specifying the initial zoom factor to use for the map. Ranges from 2 to 18. If not specified, the maximum zoom level (smallest area) that encompasses all the sites will be used.

MinZoom

Optional numeric expression, setting the lowest permitted zoom level (corresponding to the set of the largest tiles to be used, in the sense of maximum displayed area per tile).

MaxZoom

Optional numeric expression, setting the greatest permitted zoom level (corresponding to the set of the smallest tiles to be used, in the sense of minimum displayed area per tile).

Comments: The zoom factor corresponds to the tile set that

should be used within the area of the map. To zoom in or zoom out means switching to a tile set that shows a greater or lesser amount of detail, and therefore a corresponding smaller or larger area of geography.
Tiles are downloaded on request from the url specified in the property, SlippyMapRemoteTileSource1, found in Setup.INI file
If a sites array to be used, it is created by first making a call to GetSessionContainers. This returns an array of tag names to be used. This is used as the basis for a new array that contains structures for each site, build using the site's latitude, longitude and a call-back to a method for drawing the site.
The structure of the array is defined as follows:

```
{***** Structure that holds information for drawing a Site on a map *****}
MapSite Struct [
   PtrLatitude { Pointer to site's latitude in decimal degrees };
   PtrLongitude { Pointer to site's longitude in decimal degrees };
   Callback { Scope where Draw(MapObj, Lat, Lon) is called to draw marker on map};
];
```

The following example shows how this would be done.

Examples:

```
Init [
  If 1 Main;
  [
    Sites = New(2);
    SiteObj = Scope(\Code, "MySite1");
    Sites[0] = MapSite(&(SiteObj\Latitude), &(SiteObj\Longitude), SiteObj);
    SiteObj = Scope(\Code, "MySite2");
    Sites[1] = MapSite(&(SiteObj\Latitude), &(SiteObj\Longitude), SiteObj);
  ]
]
```

```
Main [
  GUITransform(0, 400, 600, 0,
               1, 1, 1, 1, 1,
               0, 0, 1, 0,
               0, 0, 0,
               \MapDraw(Invalid, Invalid, Sites));
]
```

SiteObj is expected to have a Draw module. Draw is called by MapDraw in order to draw this site at its location on the map. Draw is called with the following parameters:

MapObj

Object value of MapDraw in which various helper functions can be called.

PtrLatitude

A pointer to the site's latitude value.

PtrLongitude

A pointer to the site's longitude value.

PtrHide

A pointer to a Boolean value. When true, the result should be that Draw will hide its graphic objects.

PtrZoomLevel

A pointer to the map's current zoom level, ranging from zero to twenty where zero shows the entire world.

Draw should use the helper functions LonToX() and LatToY(), to position its graphics at the correct x,y location on the map. For example:

```
X = MapObj\LonToX(PtrLongitude);
Y = MapObj\LatToY(PtrLatitude);
```

MatchKeys

Description: Returns true if the specified keyboard keys have been pressed in the sequence given.

Returns: Boolean

Usage: Steady State only.

Function Groups: Keyboard

Related to: MakeBuff

Format: MatchKeys(Enable, Keys)

Parameters:

Enable

Required. Any numeric expression that enables the function. Testing of keyboard input is enabled when this parameter is true (i.e. not 0). If it is 0 (false), then the function's value is false. In addition, the Enable parameter controls the type of comparison done. If the Enable is 1, a case-sensitive match is made. If the Enable is 2, then the match is not case-sensitive.
(Any non-zero value other than 2 will cause a case-sensitive match. The use of 1 and 2 is recommended for clarity.)

Keys

Required. A text expression giving the key sequence to test for. The case of individual letters may be significant, depending on the Enable parameter.
To generate extended keys that are not already available as constants defined in the system layer, use MakeBuff to turn the ASCII code(s) into a text expression. For example:
PageUp = Concat(MakeBuff(1, 253), MakeBuff(1,

```
0x49))
CtrlZKey = MakeBuff(1, 26)
```

Comments:

This function should be used in a window or page module in order to monitor key strokes.

The Enable parameter is a status expression controlling the comparison. The comparison starts once the Enable becomes true. If the Enable becomes false, the function's value becomes false and the comparison starts at the beginning of the Keys string again once the Enable becomes true. This feature is useful for resetting the MatchKeys function once an action using the function's result has been performed.

The MatchKeys function is also reset automatically when it occurs in an action trigger that becomes true.

The function's result is automatically set to 0 (false) when the state containing the function is entered. Once the function becomes true, it remains true as long as the state does not change and the Enable remains true.

Any key sequence may be used for the Keys parameters including the function keys. Note that the MatchKeys function is case sensitive (upper and lower case letters are treated as different characters) when the Enable is an odd number. Often only one key is included in the Keys string. Several keys may be used in the Keys string and function as a password. The keys typed are not displayed on the screen by this function. Several MatchKeys functions may be active at any time, each comparing the keyboard input against their own Keys parameter.

Example:

```
If MatchKeys(2,"Y");
[
  ...
]
```

When the letter "Y" is typed on the keyboard, regardless of case, the action will trigger, execute its script, and reset the MatchKeys function to wait until "Y" is typed again.

Related Information:

ASCII Constants

See: "Latching and Resetting Functions" in the VTScada Programmer's Guide

Max

Description: Returns the maximum of a group of parameters.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Generic Math

Related to: AMax | AMin | Limit | Min

Format: Max(Parm1, Parm2 [, Parm3, ...])

Parameters:

Parm1, Parm2, Parm3, ...

Required. Any number of parameters giving any numeric expressions, from which a maximum value will be selected.

Comments: The order of the values is irrelevant. If any of the parameters is invalid, the return value is invalid also.

Examples:

```
p = Max(2, 3, 2.9, -3);
q = Max(3, 2);
r = Max(Invalid, 6);
```

The values of p, q, and r will be 3, 3 and Invalid respectively.

MCSInstance

Description: Module Calling Structure Instance. This function returns the object value of a module called by another module.

Warning: This function should be used by advanced users only.

Returns: Object value

Usage: Script or steady state.

Function Groups: Compilation and On-Line Modifications, Advanced Module

Related to: MCSMod

Format: MCSInstance(Code)

Parameters:

Code

Required. The code value in which the module call is embedded.

Example:

```
startedInstance = MCSInstance(mcs);
```

This makes startedInstance an object value that points to a running instance of a module started by the module given by mcs.

Related Functions:

MCSMod

Description Module Calling Structure Module. This function returns the module value from a line of code that calls that particular module.

Returns Module

Usage Script or steady state.

Function Groups Compilation and On-Line Modifications, Advanced Module

Related to: MCSInstance

Format: MCSMod(Code)

Parameters

Code

Required. The code value in which the module call is embedded.

Comments This is an advanced function for use in writing and implementing the main toolbar.

Mean

Description Returns the mean (average) of a portion of a numerical array.

Returns Numeric

Usage Script or steady state.

Function Groups Array, Generic Math

Related to: AMax | AMin | AValid | FiltHigh | FiltLow | FitOffset | FitSlope | SDev | Sum | Variance

Format: Mean(ArrayStart, N)

Parameters

ArrayStart

Required. Any numeric array element giving the starting element of the array. The index for the array may be any numeric expression and specifies the starting point for the array search. If processing a color array, the usual rules apply to decide which dimension should be examined.

N

Required. Any numeric expression giving the number of array elements to use starting at the element given by the first parameter. If N extends past the upper bound of the lowest array dimension, this computation will "wrap-around" and resume at element 0, until N elements have been processed.

Comments Invalid array elements are not included as part of the calculation, unless there are no valid numerical array elements in the specified range, in which case the function returns invalid. Invalid is also returned if either parameter is invalid, or if the number of elements to use is 0.

Example:

```
x[0] = Invalid;
x[1] = Invalid;
x[2] = Invalid;
x[3] = 1;
x[4] = 2;
x[5] = 1;
x[6] = 2;
avg = Mean(x[0], 7);
```

The value of avg will be set to 1.5.

MemIn

Description: Returns a byte, word, or longword of RAM memory.

Returns: Varies

Usage: Script or steady state.

Function Groups: Memory I/O

Related to: CopyIn | CopyOut | MemOut

Format: MemIn(Address, Type)

Parameters:

Address

Required. Any numeric expression which gives the RAM address to read. This may be specified using the

@ operator.

Type

Required. Any numeric expression giving the type of read to perform.

Type	Read Type
0	Read 8 bit byte
1	Read 16 bit word
2	Read 32 bit long word

Comments: This function requires that the VTSIO driver be installed. Please refer to the topic, Communicating Directly With Hardware for more details.
This is a high priority function. If used in a statement or action trigger, it will be evaluated at a very fast rate. MemIn should be used sparingly to avoid reduced system performance.
In certain cases, the registry may need to be modified to allow access to the memory location.

Memory

Description: Returns the amount of memory that VTScada has acquired from the OS heap for internal use.

Returns: Numeric

Usage: Script Only.

Function Groups: Memory I/O

Related to: Memory MemTrace

Format: Memory()

Parameters: None

Comments: The value returned is not the total amount of memory used by the VTScada process as it does not include other allocations that may be made from the OS heap.

Example:

```
If ! Valid(memUsed);
[
  memUsed = Memory();
]
```

MemOut

Description: Writes a byte, word, or longword of RAM memory.

Returns: Nothing

Usage: Script or steady state.

Function Groups: Memory I/O

Related to: CopyIn | CopyOut | MemIn

Format: MemOut(Address, Type, Value)

Parameters:

Address

Required. Any numeric expression that gives the RAM address to write. This may be specified using the @ operator.

Type

Required. Any numeric expression giving the type of write:

Type	Write Type
0	Write 8 bit byte
1	Write 16 bit word
2	Write 32 bit long word

Value: Required. Any numeric expression giving the value to write.

Comments: This function requires that the VTSIO driver be installed. Please refer to the topic, Communicating Directly With Hardware for more details.

In certain cases, the registry may need to be modified to allow access to the memory location.

MemTrace

Description: Writes memory allocation information to a file.

Returns: Boolean indication of success or failure.

Usage: Script Only.

Function Groups: Memory I/O functions

Related to: Memory

Format: MemTrace(Tracefile, Lowstamp, Highstamp, Lowsize, Highsize, LowID, HighID, Done)

Parameters:

Tracefile

Required. Any text expression for the name of the file to generate.

Lowstamp

Required. The low timestamp range specifier. Defaults to 0 if Invalid.

Highstamp

Required. The high timestamp range specifier. Defaults to 1E99 if Invalid.

Lowsize

Required. The low size range specifier. Defaults to 0 if Invalid.

Highsize

Required. The low size range specifier. Defaults to 0xFFFFFFFF if Invalid.

LowID

Required. The low caller id range specifier. Defaults to 0x0000 if Invalid.

HighID

Required. The high caller id range specifier. Defaults to 4096 if Invalid.

Done

Required. A return variable, which will be set TRUE after the memory trace file has been written. MemTrace is a threaded function.

Comments: If the file was generated, TRUE is returned; otherwise, FALSE is returned

Related Functions:

Merge

Description: Applies a set of changes (the output of a Diff operation) to a buffer.

Returns: Buffer

Usage: Script Only.

Function Groups: Configuration Management

Related to: Combine | Diff | Merge2

Format: Merge(SourceBuff, DiffBuff)

Parameters:

SourceBuff

Required. The buffer or stream to be modified.

DiffBuff

Required. The buffer or stream containing formatted instructions for how to modify the source buffer.

Comments: The source buffer must be identical to the origin buffer used to create the Diff buffer in the first place. This operation cannot fail, but will produce unexpected results if either input buffer is corrupt or if the wrong origin buffer is used. This function

is synchronous and returns the result of the operation.

The operations in the DiffBuff are formatted as follows:

- The low 31 bits of the first four bytes hold the length of the data portion, measured in bytes.
- The highest bit of those first four bytes will be a 0 to indicate that this is a delete operation or a 1 to indicate that this is an add operation.
- The next four bytes hold the offset into the source buffer where the operation should take place.
- The final "length" bytes (where length was defined in the first 31 bits) hold the data to be added in the case that this is an add operation.

"DiffBuff" could be the result from the Diff function. Merge assumes that the diffs in DiffBuff are ordered and correct, hence does no checking for offsets outside SourceBuff, etc. It is the responsibility of the caller to get it right.

Merge relies on "Modify" operations being ordered as an "Add" in a particular location followed by a "Delete" in the same location. This is the ordering generated by the Diff function. Reversing the order will cause undesirable results.

Examples:

Merge2

(System Library)

Description: Attempts to apply two different Diff buffers to a single origin buffer.

Returns: Result buffer.

Usage: Script Only.

Function Groups: Configuration Management

Related to: Combine | Diff | Merge

Format: \System\Merge2()

Parameters:

Source

Required. Buffer or stream to be modified .

Diff1

Required. Buffer or stream containing the first set of Diffs for modifying "SourceBuff"

Diff2

Required.Buffer or stream containing the second set of Diffs for modifying "SourceBuff"

pConflict1

Optional, pointer to a dictionary of conflicting records of Diff1

pConflict2

Optional, pointer to a dictionary of conflicting records of Diff2

pDiffStream

Optional, pointer to a stream contains all non-conflicting records from Diff1 and Diff2.

Comments: Similar to Combine, but without automatic conflict resolution or change priority.
Both Diff buffers must have started from the same origin and that origin must be the one provided. It is assumed that the Diff buffers represent different changes to the same origin.
This function fails if the changes cannot be applied cleanly, without interfering or conflicting with one

another. A conflict is defined as any of the following:

- Two additions occurring at the same location and adding different data.
- One deletion and one addition where the addition occurs within the range of the deletion.

Information describing the nature of the failure is provided to the fourth and fifth parameters. The sixth parameter is provided with a combined Diff buffer containing all of the changes that did not conflict.

Rules applied to detect conflicts:

1. Both operations delete and overlapping deletes are combined to create a composite delete. If the composite delete conflicts with a subsequent add, all of the constituent deletes of the composite delete are also marked as being conflicts.
2. Both are addition operations. Conflict can only occur at the offset.
   - If additions occur at different offsets then there is no Conflict.
   - If additions occur at the same offset with the same data, then there is no conflict.
   - If additions occur at the same offset, but adding different data then there is a conflict.
3. One operation adds and the other deletes.
   - If addition occurs within the range of the deletion operation then there is a conflict.
   - If addition occurs outside of the range of the deletion operation, then there is no conflict.

When two records are determined to be conflicting based on the previous rules, we need to go through

all previously saved records marked #Pending to move them to the conflicting dictionary.
If there are conflicts between the two Diffs, the sub-routine returns conflict information, whereas if there are none then the two diffs are simultaneously applied to the source buffer.
This operation is synchronous, the result buffer is returned by the call.

Examples:

MetaData

Note: Depending on the context in which it is used, this command has two different purposes.

Description: If used with a variable which is not a dictionary, this command attaches meta data to that variable, thereby creating a dictionary object. The primary purpose in this case is to provide a means of associating extended data with a variable.
If used with a variable which is a dictionary, this command will return the value associated with the specified key.

Returns: Varies – see description

Usage: Script or steady state.

Function Groups: Dictionary, Variable

Related to: Dictionary

Format: MetaData(dictionary, key, [case sensitive]);

Parameters:

Dictionary

Required. A variable name that will become the dictionary.

Key

Required. A text value. Integers may be used, but will be cast to text.
This will become the first key within the dictionary.

Case

An optional Boolean, controlling whether the dictionary will use case sensitive keys or non case sensitive.
TRUE (default) defines a non-case sensitive key.
FALSE defines case sensitive.

Example 1: Adding meta data to a variable

```
X  =  42;    { X is an integer variable with the value 42 }
MetaData(X, "A",  1 ) = 10;
```

X becomes a case sensitive dictionary having a root value that is the integer 42 and possessing one key, "A", that has the value 10.

Example 2: Retrieving the meta data from a dictionary

```
X  =  Dictionary(5);
X["A"]  =  42;
Y = MetaData(X, "A");
```

Y will now hold the integer 42, being the value stored with the key "A" in the dictionary X.

Min

s

Description:	Returns the minimum of a group of parameters.
Returns:	Numeric
Usage:	Script or steady state.
Function Groups:	Generic Math
Related to:	AMax | AMin| Limit | Max

Format: Min(Parm1, Parm2 [, Parm3, ...])

Parameters:

Parm1, Parm2, Parm3, ...

Required. Any number of parameters, giving any numeric expressions, from which a minimum value will be selected.

Comments: The order of the values is irrelevant. If any of the parameters is invalid, the return value is invalid also.

Examples:

```
p = Min(62, 3, 2, 2.01);
q = Min(3, 2);
r = Min(Invalid, 6);
```

The values of p, q, and r will be 2, 2 and invalid respectively.

MkDir

Description: Create a new folder (directory) and returns its own error code.

Returns: Numeric

Usage: Script Only.

Function Groups: File I/O

Related to: RmDir

Format: MkDir(Name)

Parameters:

Name

Required. Any text expression which is the full path name of the directory to create. A known path Known Path Aliases for File-Related Functions for File-Related Functions may be provided in the form, : {KnownPathAlias}.

Comments: The return value is as follows

Return Value	Meaning
0	Directory successfully created
1	Directory already exists
2	Creation failed
3	Bad path

In VTS version 7.0 and later, MkDir can now create directories recursively, so that

```
MkDir("C:\one\two\three");
```

will create directory one (if it does not already exist), then create directory two (if it does not already exist), and then finally create directory three (if it does not already exist).
If Name is given as a relative path, it will be created below the VTScada installation directory.

Example 1:

```
  err = MkDir("C:\SAMPLE");
```

This creates the directory Sample on drive C.

Example 2:

```
err = MkDir("SAMPLE");
```

This creates the directory C:\VTScada\SAMPLE

ModemCount

Description Returns the number of data modems configured and operational in the system.

Returns Numeric

Usage Script or steady state.

Function Groups Modem

Related to: ModemDial | ModemList | ModemMedia | ModemStream | ModemTransfer

Format: ModemCount()

Parameters None

ModemDev

Description: Obtains the identifier for a modem sub-device. (In order to use the audio capabilities of a voice modem, the device identifier for the wave output device is required.)

Returns: See description

Usage: Script Only.

Function Groups: Modem

Related to: ModemMedia | Sound

Format: ModemDev(ModemStream, DeviceName)

Parameters:

ModemStream

Required. The connected modem stream (returned from a ModemStream or ModemDial function call).

DeviceName

Required. A text value describing the device whose identifier is required. This should be "wave/out" to obtain the identifier of the audio output device.

Comments: When called with a DeviceName of "wave/out", this function returns a value suitable for use with the SpeechSelect and Sound functions.
Modem audio devices typically require 8-bit mono audio at 8000 samples/sec. If the audio to be played does not conform to this format, then the sound will not be played. Speech engines usually have special voices available for

telephony applications – see the SpeechEnum function.

ModemDial

Description: Dials and attempts to connect to a remote modem and returns the modem stream or an error code.

Returns: Modem Stream or Error Code

Usage: Script Only.

Function Groups: Modem

Related to: ModemCount | ModemList | ModemMedia | ModemStream | ModemTransfer

Format: ModemDial(Modem, PhoneNumber, ReceiveLen, Trans-mitLen, MinBaud, MaxBaud, DataBits, StopBits, Parity, XonXOff [, MediaMode, Timeout, InitString])

Parameters:

Modem

Required. Either a numeric expression giving the modem number between 1 and the number of modems in the system (the return from ModemCount), or a modem stream opened via the ModemStream function.

PhoneNumber

Required. Any text expression specifying the phone number to dial. It can contain any of the special dialing characters that WindowsÔ allows. If this parameter is invalid a pass-through connection will be obtained.

ReceiveLen

Required. Any numeric expression giving the size of the receive buffer for the serial stream in bytes. ReceiveLen must be in the range 2 to 32 766. If more bytes are received than can fit in the receive buffer before the application removes them, the additional

data will be lost.

TransmitLen

Required. Any numeric expression giving the size of the transmit buffer for the serial stream in bytes. TransmitLen must be in the range 2 to 32 766. The buffer must be large enough to hold the maximum number of bytes pending transmission at any instance.

MinBaud

Required. Any numeric expression giving the minimum baud rate which will be acceptable on the modem connection. The baud rate at which the connection is made may actually be higher than this value but may not exceed MaxBaud. The baud rate must be in the range 0 to 115,200, and must divide evenly into 115,200 with no more than 2.5% error. The value of 0 has special significance. This corresponds to the maximum baud rate available for the particular device.

MaxBaud

Required. Any numeric expression giving the maximum baud rate which will be acceptable on the modem connection. The baud rate at which the connection is made may actually be lower than this value but may not be less than MinBaud. The baud rate must be in the range 0 to 115,200, and must divide evenly into 115,200 with no more than 2.5% error. The value of 0 has special significance. This corresponds to the maximum baud rate available for the particular device.

DataBits

Required. Any numeric expression giving the number of data bits per character. DataBits must be 5, 6, 7, or 8.

StopBits

Required. Any numeric expression giving the number

of stop bits per character. StopBits must be 1 or 2.

Parity

Required. Any numeric expression giving the parity checking to use:

Parity	Checking Type
0	No parity
1	dd parity
2	Even parity
3	Stick (space parity)
4	1 Stick (mark parity)

XOnXOff

Required. Any logical expression. If true (non-0) software flow control is to be used. If false (0) it is not. Since software flow control is only supported for DigiBoards, this should normally be set to 0.

MediaMode

An optional numeric expression, specifying the desired media mode for the call. See ModemMedia for a description of valid values and their meanings.

Timeout

An optional parameter that should be set to the number of seconds after the completion of dialing that the call should be allowed to wait before it is automatically abandoned as NOANSWER. A value of 0 indicates that the application does not desire automatic call abandonment. The default value of Timeout is 0.

InitString

An optional text value that will be sent to the modem as the final part of initializing it for use, just before it dials out. The string should include any terminating characters that the modem will require to complete the

string, such as "\r".

Com-
ments:

Return	Status
0	Idle (no connection), waiting for a call
1	Starting outbound call
2	Outgoing call request accepted by modem
3	Dial tone on outbound call
4	Dialing outbound call
5	Remote phone ringing on outbound call
6	Remote phone busy on outbound call
7	Connected (should not be seen since a stream value will be returned at this point)
8	Another application is handling the call after a ModemTransfer
9	Incoming call detected. The modem is being called, but has not yet answered. Changes to 11 when the modem answers.
10	The modem is temporarily unavailable. Occurs when trying to place an outgoing call and either the modem is still initializing or another TAPI aware application has the modem.
11	The modem has answered a call. Modem training in progress. Normally followed by a valid stream connection. If an error has occurred, the return values will be one of the following error codes:

Error code	Meaning
-1	The (numeric) first parameter to the function is out of range
-2	No dial tone detected on outbound call
-3	Remote phone busy on outbound call
-4	No answer on outbound call

ModemDigits

Description: Controls the receipt of DTMF digits entered from the keypad of a telephone engaged on a voice call via a voice modem.

Returns: Nothing

Usage: Script Only.

Function Groups: Modem

Related to: GetStreamLength | ModemDial | ModemMedia | ModemStream | SRead

Format: ModemDigits(ModemStream, Enable)

Parameters:

ModemStream

Required. The connected modem stream (returned from a ModemStream or ModemDial function call).

Enable

Required. Any logical value that is "true" (non-zero) to enable digit monitoring, and "false" (zero) to cancel digit monitoring.

Comments: Once digit monitoring is enabled any digits detected by the modem will appear as text characters (0123456789#*) in the ModemStream. The presence of these characters can be determined by using the GetStreamLength function and they can be read using the SRead function.

ModemList

Description: Returns a list of the modems in the system. This function should be used to determine the correct parameter for the ModemStream or ModemDial functions.

Returns: Array

Usage: Script only.

May be used in optimized Tag Parameter Expressions.

Function Groups: Modem

Related to: ModemCount | ModemDial | ModemMedia | ModemStream | ModemTransfer

Format: ModemList()

Parameters: None

Comments: The return value is a two-dimensional array with each row storing the information for a single modem (i.e. ArraySize(ReturnValue, 0) = = ModemCount( )). There are four columns per row as follows

Column #	Information
0	The operating system enumeration for this device
1	Comport identifier (e.g. "COM1")
2	'Friendly name" of the modem (e.g. "Standard Modem")
3	MediaMode. Bit significant value indicating the capabilities of the modem. Please refer to ModemMedia for a description of valid values and their meanings.
4	If set to1, the modem is controlled by the Trihedral TSP. The Trihedral TSP is to be preferred over other options.

The Microsoft Telephony Interface (TAPI) enumerates a number of devices - not just modems. This function enumerates only the data/voice modems into a compact list. The ordering of this list should not be relied upon as it may change due to system hardware changes.

To use this function:

1. Call the ModemList function
2. Iterate through the resulting array, choosing a modem by its 'Friendly Name' or by its 'MediaMode' capabilities.
3. Add one to the array index and use the resulting value as the first parameter to ModemStream or ModemDial.

ModemMedia

Description: Enables you to determine the media mode of a serial stream open on a modem, and change it if necessary (for example, if you require the ability to be able to handle both incoming voice mode and data mode calls).

Returns: Numeric

Usage: Script Only.

Function Groups: Modem

Related to: ModemDev | ModemDial | ModemDigits | ModemList | ModemStream | ModemTransfer | Sound

Format: ModemMedia(ModemStream [, MediaMode])

Parameters:

ModemStream

Required. The connected modem stream (returned from a ModemStream function call) that is to be handed off to another interested application.

MediaMode

Required. Specifies the desired media mode for the call. Valid values are bit significant, and more than one may be specified.

MediaMode	Meaning	Comments
2	Unknown	Is set whenever more than one of the other values is set.
4	Interactive Voice	A voice mode call using the modem microphone/speaker.
8	Automated Voice	For simulated speech and wave files. Allows DTMF digit detection.
16	Datamodem	For data calls.
32	Group 3 Fax	For FAX calls.

Comments:

MediaMode is important when handling multi-mode calls. It may be specified when dialing a call (ModemDial) or preparing a modem for answering incoming calls (ModemStream), and is a consideration before handing off a call to another service (ModemTransfer). If more than one MediaMode bit is set, then the call type is undetermined, and Unknown (2) should also be set.
It is not possible to change the media mode to a mode other than those specified when the ModemStream was created. If the requested Media change cannot be implemented, the function will return Invalid. The function will also return Invalid if there is no active call in progress on ModemStream.

If Interactive Voice (4), or Automated Voice (8) is specified, then the modem will be initialized in voice mode. On out-

going calls, this has the effect that progress indication is unreliable – the call will be reported as connected as soon as dialing completes. This is a limitation of analog voice modems.

Changes to media mode may not happen instantly. For example, changing a call from voice mode to data mode initiates the modem training sequence that may take several seconds to complete. The return value of the ModemMedia function indicates the current state, and so can be used to determine when the change has completed.

Example:

As an example, the following steps would be required to provide an automated alarm reporting system that hands off incoming data calls to the system RAS service.

1. Enumerate the available modems with ModemList and choose one with DataModem and Automated Voice facilities.
2. Use ModemStream to prepare the modem for incoming calls, specifying Datamodem, Automated Voice, and Unknown as the media mode.
3. When a call arrives, it will initially be answered in voice mode. Use ModemDigits to set up digit monitoring.
4. Use ModemDev to obtain the handle for the wave/out device. This handle can then be used with the Sound function to play a wave file, or with the SpeechSpeak function to play simulated speech. A typical initial function would be to request that the caller press a digit key on the phone.
5. If, for example, no DTMF digit is received within a few seconds, then it may be assumed that this is actually a data call. In this case, ModemMedia should be called to set the Datamodem media mode. If the return value is valid, then the mode should be polled approximately every 0.5 second, until ModemMedia returns a value that no longer contains Automated Voice. The ModemTransfer function may now be called to transfer the call, and the modem stream can be closed.

ModemStream

Description: Open a serial stream on a modem and returns its status (prior to the connection being made), a modem stream (after the connection has been established), or an error code.

Returns: Varies – see description

Usage: Script Only.

Function Groups: Modem

Related to: ModemCount | ModemDial | ModemList | ModemMedia | ModemTransfer

Format: ModemStream(Modem [, RingCount, ReceiveLen, TransmitLen, MediaMode])

Parameters:

Modem

Required. Any numeric expression giving the modem number between 1 and the number of modems in the system (the return from ModemCount).

RingCount

An optional parameter that is any numeric expression designating the number of rings after which the modem should be answered. If this parameter is 0 the modem will not answer the call. The default is 1.

ReceiveLen

An optional parameter that is any numeric expression giving the size of the receive buffer for the serial stream in bytes. ReceiveLen must be in the range 2 to 32 766. The default length is 1024. If more bytes are received than can fit in the receive buffer before the application removes them, the additional data will be lost.

TransmitLen

An optional parameter that is any numeric expression

giving the size of the transmit buffer for the serial stream in bytes. TransmitLen must be in the range 2 to 32 766. The default length is 1024. The buffer must be large enough to hold the maximum number of bytes pending transmission at any instance.

MediaMode

An optional numeric expression, specifying the desired media mode for the call. See ModemMedia for a description of valid values and their meanings.

Comments

Typically, this function is used to prepare a modem for the receipt of incoming calls. Use the ModemDial function to make outgoing calls. The return value for this function will be one of the following integer values until the modem has a valid connection with a remote modem

Return Value	Status
0	Idle (no connection), waiting for a call
1	Starting outbound call
2	Outgoing call request accepted by modem
3	Dial tone on outbound call
4	Dialing outbound call
5	Remote phone ringing on outbound call
6	Remote phone busy on outbound call
7	Connected (should not be seen since a stream value will be returned at this point)
8	Another application is handling the call after a ModemTransfer
9	Incoming call detected. The modem is being called, but has not yet answered. Changes to 11 when the modem answers
10	The modem is temporarily unavailable. Occurs when trying to place an outgoing call and either the modem is still initializing or another TAPI aware application has the modem
11	The modem has answered a call. Modem training in progress. Normally followed by a valid stream connection

If an error has occurred, the return values will be one of the following error codes:

Error Code	Error
-1	The (numeric) first parameter to the function is out of range

ModemTransfer

Description: Transfers a modem call to another application and returns an indication of success.

Returns: Numeric

Usage: Script Only.

Function Groups: Modem

Related to: ModemCount | ModemDial | ModemList | ModemMedia | ModemStream

Format: ModemTransfer(ModemStream)

Parameters:

ModemStream

Required. The connected modem stream (returned from a ModemStream function call) that is to be handed off to another interested application.

Comments: The call is transferred to the next highest priority application interested in the call. When the transfer is made, the stream value will be set to 8. This function will return true (1) to indicate a successful transfer or false (0) if the transfer was unsuccessful.
Passing an incoming call to a RAS would be a common use for this function.

ModifyAccount

Security Manager Module

Description: Used to change any of the elements of an account definition that may be modified.

Returns: Object value

Usage: Script Only.

Related to: AddAccount | DeleteAccount

Format: \SecurityManager\ModifyAccount (NewAccountData [, PtrReturnCode, HaveLock]);

Parameters:

NewAccountData

Required. An AccountData structure, a single dimension array of AccountData structures or a dictionary of AccountData structures containing the data to modify in each account.

PtrReturnCode

Optional. A pointer to a value that will contain one of the defined result codes at the conclusion of the operation.

HaveLock

Optional. A Boolean value that indicates whether the working copy lock is held by the calling code. Default FALSE.

Comments:

To use this API, the calling code must be running in a security session that has Manager privilege.

Modifying an account is an asynchronous operation. If the asynchronous operation was not attempted, due to detection of an error, the return value will be Invalid. If the asynchronous operation is attempted, the return value will be an object value. The object value will become Invalid when the asynchronous operation completes. At that time (or when the method returns Invalid), the value addressed by PtrReturnCode can be examined to determine the status of the operation. The contents of the value addressed by PtrReturnCode is undefined until the method returns Invalid.

A single account can be modified by supplying a single AccountData structure in NewAccountData. Multiple accounts can be modified in one operation by providing a

single dimension array or dictionary of AccountData structures in NewAccountData.
The result code returned in the value addressed by PtrReturnCode will be a scalar value if a single structure was supplied in NewAccountData. If an array of structures or a dictionary of structures was supplied, a single dimension array of the same size as NewAccountData will be returned in the value addressed by PtrReturnCode, each element containing the result code for the corresponding NewAccountData element.
Modifying an account requires a working copy write lock. If such a lock is held by the calling code, the HaveLock parameter must be set to TRUE. Otherwise omit this parameter or set it to FALSE. If the calling code holds a read lock on the working copy, this must be released before ModifyAccount can complete its operation.
The AccountData structure(s) provided must have the AccountID member set to an existing account ID.
Any other member of the structure can be Invalid, in which case no change is made to that member of the account record. Only valid members cause modification.
If the password is being changed, the new password must be conformant with application password strength settings. On return the Password member is not erased. It is highly recommended that calling code be careful to ensure that unencrypted passwords are destroyed as soon as possible after completion of this operation.

ModifyBitmap

Description: Reads an existing image handle and produces a new one with modifications. The original image is not altered. Returns invalid upon failure.

Returns: Image

Function Groups: Graphics

Usage: Script or steady state.

Related to: Crop | GUIBitmap | GUIButton | ImageArray | ImageSweep | MakeBitmap

Format: ModifyBitmap(Handle[, Reflect, Hue, Saturation, Lightness, Transparency, Contrast, ColorizeHue, ColorizeSaturation, AntiAlias, ScalarColor, Rotation])

Parameters:

Handle

Required. The image handle to copy and modify.

Reflect

An optional logical value that can be set to 1 to reflect the image about the vertical axis. The default is no reflection (0).

Hue

An optional numeric expression for the hue rotation to apply to all of the colors in the image. Range -180 to 180. The default is no hue rotation, (0).

Saturation

An optional numeric expression that is a multiplier to the intensity of each primary color component of every color in the image. Range: 0+. Defaults to 1 if missing or invalid.

Lightness

An optional numeric expression that is a multiplier to the brightness of the image. Range 0+. Defaults to 1 if missing or invalid.

Transparency

An optional numeric expression for a multiplier to the transparency (alpha value) of every color in the image. Range 0 (transparent) to 1 (opaque). Defaults to 1 if missing or invalid.

Contrast

Optional numeric expression to enhance the differences between the colors in the image. Range 0+. Defaults to 1 if missing or invalid.

ColorizeHue

Optional numeric expression for the hue value of a color to be mixed with every color in the image. Range -180 to 180. Defaults to 0 if missing or invalid.

ColorizeSaturation

Optional numeric expression for the intensity of the added color. Zero means that no color is added. Range 0-1. Defaults to 0 if missing or invalid.

AntiAlias

Bit 0 controls whether anti-aliasing will be done when the image is stretched. Defaults to 1 (TRUE) if missing or invalid.
By default, feathering will be applied to an anti-aliased image when it is stretched. Set bit 1 to 1 (TRUE) to suppress feathering.

ScalarColor

Optional color value in the form aRGB[1], used to apply a specific color to a gray-scale image.

Rotation

Optional floating point value, specifying the clockwise rotation in degrees to be applied when the image is drawn.

Comments

Some image modifications have a greater performance impact than would typically be assumed, due to the way

---

[1]A colour value, defined as four, two-digit hexadecimal values. Alpha - Red - Green - Blue. An Alpha value of FF is assumed if only the RGB values are provided.

image rendering is optimized. Scaling and cropping changes are slow, for example

ModifyConfiguration

Description: Provides a safe way to write to configuration files.

Returns: Nothing

Usage: Script Only.

Function Groups: File I/O

Related to: FRead | GetUserID | ReadConfiguration | WritePropertiesFile

Format: ModifyConfiguration(pSuccess, CallBackModuleName[, ExtraInfo, Deploy])

Parameters:

pSuccess

A required pointer to a value, which will be set to 1 on successful completion or 0 on failure.

CallBackModuleName

A required text value, containing the name of the call-back module to be launched into the caller.

ExtraInfo

An optional parameter which may be any value. If present, it will be passed through to the callback module.

Deploy

Optional Boolean. Has no effect while the application is running in automatic-deploy mode. If auto-deploy is off, then when this parameter is TRUE (or Invalid) the modified files will be deployed immediately after the changes occur. Defaults to TRUE.
Deploy must be present and set to FALSE to prevent ModifyConfiguration from acting as if auto-deploy is

always on.

Comments: If the caller of ModifyConfiguration is subscribed to working copy changes (via WCSubscribe), then it will not get subscription callbacks for its own changes following a call to this function.
The callback module, named in the second parameter, must be either a launched module or a subroutine that returns Invalid. The callback module is allowed to write configuration files and is guaranteed that, for the life of the module, no other configuration code can modify that file.

The callback module must have the following parameters

pUserID

A pointer variable, to which the current user ID should be assigned. This will be saved as part of the version history.

pComment

A pointer variable for a text comment, to be recorded in the version history.

ChangedFiles

A dictionary that must be populated with the names of the files to be changed during the callback. For each file, add an entry with the key being the file name and the value set to zero. File names may be absolute or relative.
The callback must record the changed files in this dictionary or else they will not be recorded in the configuration management system or deployed to other workstations.

ExtraInfo

Extra data to be passed to the callback function.

Example:

```
  If 1 WaitWritten;
  [
 \ModifyConfiguration(&Success, "WriteConfig", Invalid, FALSE);
  ]
…
]

<
{============================== \WriteConfig
==============================}
{=========================================================================-
=====}
WriteConfig
 (
  pUserID                   { User that gets recorded with change
};
  pComment                  { Comment that gets recorded with change
};
  ChangedFiles              { Dictionary to populate with names of
changed
                              files
};
  ExtraInfo                 { Accepts whatever was in the third para-
meter
                              of
ModifyConfiguration                       };
)

Main [
  If 1;
  [
    *pUserID  = \GetUserID();
    *pComment = "Updating MyService configuration";
    ChangedFiles["MyConfigFile.txt"] = 0;
     { The first line of the file is written from Info1,
}
     { and the second line is written from Info2.
}
    FWrite("MyConfigFile.txt" { file name        },
           1                  { clear file        },
           0                  { offset            },
          "%s\r\n%s\r\n"      { format string     },
          Info1, Info2        { values to write   });
    Return(Invalid);
  ]
]

{ End of WriteMyServiceConfig\WriteConfig }
>
```

ModifyTags

Description: Can be used to create, modify, or delete running tags. Replacement for StartTag for the case where persisted tags were created. See comments for more detail.

Returns: Object reference

Usage: Script Only.

Function Groups: Configuration

Related to:

Format: ModifyTags(pSuccess, TagParameters[, newTags, UserID, Comment, Merge, HaveLock, pErrors)

Parameters:

pSuccess

An optional pointer to a value, which will be set to 1 on successful completion or 0 on failure.

TagParameters

Dictionary of tag parameters, each of which is also a dictionary. See examples.

NewTags

Optional. Dictionary of tag additions. Key is tag friendly name and value is tag type.
If you have a NewTags entry for a tag that already exists, then the action will be a deletion and an add of that tag. The added tag will not have the same UniqueID as the original tag, and therefore page references and history for the old tag will not be associated with the new tag,

UserID

Optional. User making change

Comment

Optional. Comment to store with changes

Merge

Optional. On update, true to retain parameters not specified in parameter dictionary, false to revert all other parameters to their defaults. Defaults to TRUE

HaveLock

Optional. If false, this module gets the working copy lock and commits changes and update tags when finished. If true, caller must have WC lock ModifyTags will not get WC lock, commit changes, or update running tags. Defaults to FALSE

pErrors

Optional output: dictionary of errors, keyed by name of tag with error, containing #MODTAGS_* error code. Defined in \Code.
Constants used as error codes, returned in the pErrors dictionary parameter of ModifyTags:

Constant	Value	Description
#MODTAGS_BADNAME	1	Name of added tag is not allowed. Refer to tag naming rules.
#MODTAGS_BADTYPE	2	Provided type is not an existing type
#MODTAGS_MISSINGTAG	3	Attempting to modify a non-existent tag.
#MODTAGS_MISSINGPARENT	4	Attempting to add a child to a non-existing parent.

Constants used as global error codes returned in root value of ModifyTags's pErrors dictionary parameter:

Constant	Value	Description

#MODTAGS_NOERROR	0	No error.
#MODTAGS_TAGERROR	1	One of the errors listed in the preceding table.
#MODTAGS_NOTEDITABLE	2	Application cannot be edited at this time. (Most likely, a restart is required.)

Comments: Some readers may have used non-persisted StartTag calls to create parent/child tag structures. Now that StartTag is obsolete, those structures should be created with the Tag Browser, not in code.
This module launches a worker module into \Code so that the operation is not interrupted by this module's caller being slain. The return value of the function is a reference to this worker module. By watching for this to become invalid, you can discover when the module has finished.

ModifyTags works by modifying root tag declarations or adding overrides for child tags not defined in the root tag database. It does not modify tag type definitions.

The TagParameters dictionary is formatted as follows:

```
TagParameters[FullFriendlyName] = {TheChange};
```

- For tag deletion, {TheChange} is simply Invalid.
- For a creation or update, {TheChange} is a dictionary of parameter values, keyed by parameter name.

When creating a new tag, the NewTags dictionary must have an entry specifying the tag type. If there is a tag with the given name already running, it will be deleted and the new tag will be created with a new unique ID, even if the type specified in NewTags is the same as the type of the currently-running tag. Hence, this is a way to stop a tag and create a tag having the same friendly name, but a different unique ID and possibly a different type, all in a single operation.

When updating a tag, what will happen to tag parameters that are not specified in the TagParameters dictionary will depend on the value of the Merge parameter. If TRUE (the default), then those parameters remain unchanged by the operation. If FALSE, then those parameters revert to their default, where default for a root tag is the default specified in the tag type, and the default for a child tag is the value specified by its parent's type's child tag definition record (that is, the value it would have without any root overrides). The Merge parameter has no effect on tag additions or deletions.

If HaveLock is TRUE, then when this module finishes, the memory cache of tag parameters will have been updated, and those changes will be pending to be written out to tag files and incorporated into running tags. Note that there is no guarantee that those actions have been performed. That will happen during the next commit. If HaveLock is FALSE, then it is guaranteed that all running tags affected

by these changes will have been updated or stopped (whichever applies). Also, while all root tags and all child tags (whose parents are running and that the changes cause to bring into existence) will be running, it is not guaranteed that they will have run their Refresh modules. It is also not guaranteed that children of tags just started will be running yet let alone Refreshed.

Examples:

Example 1 – Modify running tags (A and B exist and are running):

```
TagParms = Dictionary();
NewTags  = Dictionary();

TagParms["A"] = Dictionary();
TagParms["A"]["Description"] = "Updated description1";
TagParms["A\B"] = Dictionary();
TagParms["A\B"]["Description"] = "Updated description2";

Success = Invalid;
Code\ModifyTags(&Success, TagParms, NewTags, GetUserID(), "Modify
tags", TRUE, FALSE, &Errors);
```

Example 2 – Add child tags (Tag C exists and is running):

```
TagParms = Dictionary();
NewTags  = Dictionary();

NewTags["C\D"]  = "AnalogInput";
TagParms["C\D"] = Dictionary();
TagParms["C\D"]["Description"] = "New AI description";
NewTags["C\D\E"] = "DigitalInput";
TagParms["C\D\E"] = Dictionary();
TagParms["C\D\E"]["Description"] = "New DI description";

Success = Invalid;
Code\ModifyTags(&Success, TagParms, NewTags, GetUserID(), "Add tags",
TRUE, FALSE, &Errors);
```

Example 3 – Delete a tag (Tag C\F exists. The third parameter must be invalid. The sixth must be false.):

```
TagParms = Dictionary();
TagParms["C\F"] = Invalid;
Success = Invalid;
Code\ModifyTags(&Success, TagParms, Invalid, GetUserID(), "Delete
Tags", FALSE, FALSE, &Errors);
```

Example 4 - Add a parent and a child tag (G and H do not exist):

```
TagParms = Dictionary();
NewTags  = Dictionary();

NewTags["G"]  = "Calculation";
TagParms["G"] = Dictionary();
TagParms["G"]["Description"] = "New Calc description";
NewTags["G\H"]  = "AnalogStatus";
TagParms["G\H"] = Dictionary();
TagParms["G\H"]["Description"] = "New AS description";

Success = Invalid;
Code\ModifyTags(&Success, TagParms, NewTags, GetUserID(), "Add tags",
TRUE, FALSE, &Errors);
```

After ModifyTags completes, the following changes to the running tags are guaranteed:

- A and A\B will be updated.
- C\F will be stopped.
- C\D and G will be running but not necessarily refreshed yet.
- C\D\E and G\H (added tags whose parents weren't already running) will not necessarily be running yet.

If TagParms includes the deletion of a parent tag, then all TagParms and NewTags records related to the addition/modification/deletion of any children of that tag are discarded.

ModifyUserPrivilege

(Security Manager Library)

Description:	Modifies a privilege for the specified username.
Returns:	Numeric (via the first parameter)
Usage:	Script Only.
Function Groups:	Security
Related to:	
Format:	\SecurityManager\ModifyUserPrivilege(PtrReturnCode, Username, Privilege, SetPrivilege[, HaveLock])
Parameters:	

PtrReturnCode

Required. A pointer to a variable that will be used for the return code.

PtrReturnCode	Meaning
1	Privilege modified.
2	Denied. The calling context does not have the Manager system privilege.
3	The privilege is not valid – no action taken.
4	The specified user does not exist – no action taken.
5	The SetPrivilege flag is invalid – no action taken.
6	The application cannot be edited.

UserName

Required. Any expression for the name of the user account to modify.

Privilege

Required. Any numeric expression for the privilege to be modified. Use a negative value for a system privilege and a positive value for an application privilege.

SetPrivilege

A Boolean expression. Set TRUE if the privilege is to be added.

HaveLock

Optional Boolean expression. Set to true if we have the WC lock. Defaults to 0 or FALSE.

Comments: May only be called from a user-context that has the Man-

ager system privilege. The return value of the function is the object value of the launched worker module. This will be set to Invalid when the operation has completed and may be used to discover when that occurs.
Use of this function requires an understanding of the VTScada security system and the system privileges. Please refer to System Privileges in the chapter Security Manager Service.

ModuleFileName

Description: Returns the full path (including the drive letter) and file name of the document (.SRC) file of a module.

Returns: Text

Usage: Script or steady state.

Function Groups: Compilation and On-Line Modifications, File I/O, Advanced Module

Related to:

Format: ModuleFileName(Module)

Parameters:

Module

Required. Any expression for the module to enquire about.

Example:

```
If 1 Next;
[
  PageFiles = ChildDocs(Scope(\Code, PageName), 10);
  I = 0;
  WhileLoop(I < ArraySize(PageFiles, 0),
          PageFiles[I] = ModuleFileName(PageFiles[I]);
           I++;
  );
]
```

ModuleHighlighted

Note: Deprecated. Do not use in new code.

Description: Returns true if the module is highlighted.

Returns: Boolean

Usage: Script or steady state.

Function Groups: Module Tree Diagram

Related to: HighlightModule

Format: ModuleHighlighted(ModuleTree, Module)

Parameters:

ModuleTree

Required. Any expression for the module tree value.

Module

Required. Any expression for the module.

Month

Description: Returns the month for a given date number.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Time and Date

Related to: Date | DateNum | Day | Today | Year

Format: Month(Date)

Parameters:

Date

Required. Any numeric expression giving the number of days since January 1, 1970.

Comments: This function works in conjunction with the Day and Year functions to decompose a date number into the cor-

responding day, month and year. January is month 1.

Example:

```
myMonth = Month(8394 { 25 December 1992 });
```

The variable myMonth will be given the value 12.

MoveEditor

Note: Deprecated. Do not use in new code.

Description Moves the Editor to the given line and column.

Returns Nothing

Usage Script or steady state.

Function Groups Editor

Related to: AddEditorText | Editor | MakeEditor

Format: MoveEditor(EditorVal, Line, Column)

Parameters

EditorVal

Required. An editor Value which is returned by MakeEditor.

Line

Required. Any numeric expression that specifies the line to move the editor to.

Column

Required. Any numeric expression that specifies the column to move the editor to.

Example:

```
aNewEditor = MakeEditor();
...
MoveEditor(aNewEditor, 22, 1);
```

This statement will cause the cursor to move to line 22, column 1 of aNewEditor.

MoveSibling

Note: Deprecated. Do not use in new code.

Description: Moves the position of a module in a module tree diagram.

Warning: This statement should be used by advanced users only since irrevocable alteration of your application may occur.

Returns: Nothing

Usage: Script Only.

Function Groups: Module Tree Diagram

Related to: MoveSelState | MoveState

Format: MoveSibling(Source, Destination)

Parameters:

Source

Required. Any expression for the module to be moved.

Destination

Required. Any expression for the module code value of the sibling module where Source will be moved.

Comments: Source will be moved to the position occupied by Destination. Source and Destination must have exactly the same parent module.

MoveWindow

Description: Will move a window to the specified coordinates.

Returns: Nothing

Usage: Steady State only.

Function Groups: Graphics, Window

Related to: CurrentWindow | SizeWindow | WindowOptions | Window

Format: MoveWindow(Win, X, Y)

Parameters:

Win

Required. Any expression that gives an object value contained in the window to move.

X

Required. Any numeric expression giving the pixel coordinate to place the left hand side of the window.

Y

Required. Any numeric expression giving the pixel coordinate to place the top of the window.

Comments: The pixel coordinates given are relative to the upper left hand corner of the screen if this is a non-child window. If this is a child window, then the coordinates are relative to the top left corner of the virtual client area of the parent window. This is consistent with the Window function.
The corner being point (0,0). The X coordinates increase as you move right across the screen. The Y coordinates increase as you move towards the bottom of the screen.

Example:

```
MoveWindow(CurrentWindow(), 20, 20);
```

This statement will cause the current window to move to the upper left corner of the screen. (Will not cause an error in script mode, but this is not common usage)

MuteSound

(Alarm Manager module)

Description This subroutine is used to turn off alarms sounds for all alarms, both current and future.

Returns Numeric

Usage Script

Function Groups Alarm, Speech and Sound

Related to:

Format: \AlarmManager\MuteSound([Invalid, MuteState, ExpiryTime]);

Parameters None.

Invalid

Placeholder. This parameter is now obsolete.

MuteState

Optional Boolean. Set TRUE to mute sound or FALSE to unmute. Defaults to TRUE.

ExpiryTime

Optional numeric. Time at which a muted alarm should unmute (UTC). Leave empty to toggle alarm muting immediately according to the MuteState setting.

Comments This subroutine will check the current user session to ensure that the logged-on user has the required privilege to toggle alarm muting before proceeding.

Note: The alarm Mute button can be selected or released for each individual user or for computers in the network, according to application configuration. See: ApplyMuteSilencePerComputer, and ApplyMuteSilencePerUser in the Manager's Guide.

Example:

```
IF AlarmShouldBeMuted;
[
  \AlarmManager\MuteSound();
]
```

N Functions

The sections that follow identify all VTScada functions beginning with "N".

New

Description	Allocates memory for an array from RAM and returns a pointer to that array.
Returns	Pointer
Usage	Script Only.
Function Groups	Array, Memory I/O
Related to:	AddVariable | AdjustArray
Format:	New([Dimensions, Start], Size) **{ Mode 1 }** Or New(FirstDimension, SecondDimension) **{ Mode 2 }**
Parameters	

{ Mode 1 }

Dimensions

An optional parameter that is any numeric expression giving the number of array dimensions to allocate. To allocate a simple value, use 0, and for a one dimensional array, use 1.
If this parameter is omitted and the function has only 1 parameter, a single dimensional array is created. If it is omitted and the function has 2 parameters (see FirstDi-

mension and SecondDimension), a two dimensional array is created.

Start

An optional parameter that is either a numeric expression, or an array.
If Dimensions is omitted, this parameter must also be omitted.
If Dimensions is 0, this is ignored. If Dimensions is 1, this is treated as a number, which is the index of the first element in the array allocated.
If Dimensions is greater than 1, this is must be the first element of an array of numbers, each element indicating the starting index for a dimension.

Size

Required. Either a numeric expression or an array. If Dimensions and Start are omitted, a single value will define the number of elements in the single dimension array that is created.
If Dimensions is 0, this parameter is ignored. If Dimensions is 1, this is treated as a number, which is the number of elements in the array allocated. If Dimensions is greater than 1, this must be the first element of an array of numbers, each element indicating the number of elements in a dimension.

{ Mode 2 }

FirstDimension

Required. A numeric value that determines the number of elements in the first dimension of the two dimensional array that is created.

SecondDimension

Required. A numeric value that determines the number of elements in the second dimension of the two dimensional array that is created.

Comments The New function is limited to a maximum of 2,000,000 elements for each dimension.

Examples:

```
If 1 NextState;
[
  simple = New(0, 0, 0); { Creates a simple value }
  array1Ptr = New(1, 1, 10);{ Creates a 1-dimensional array }
  start[0] = 1; { Dimension 1 - Starting index }
  start[1] = 2; { Dimension 2 - Starting index }
  length[0] = 3; { Dimension 1 - Number of elements }
  length[1] = 4; { Dimension 2 - Number of elements }
  array2Ptr = New(2, start[0], length[0]);
  { Creates a 2-dimensional array }
]
```

The above script allocates memory for 3 variables:
simple is a simple value
array1Ptr is a 1-dimensional array with 10 elements, numbering from 1 to 10
array2Ptr is a 2-dimensional array having 3 elements in its first dimension, numbering from 1 to 3, and 4 elements in its second dimension, numbering from 2 to 5.
array1Ptr could also have been created with the same attributes by using the following statement:

```
array1Ptr = New(10);
```

array2Ptr could have been created with same number of rows and columns, but whose indices started from 0 by using:

```
array2Ptr = New(3, 4);
```

NextFocusID

Description: Moves the focus position to a specific ID number.

Returns: Nothing

Usage: Script Only.

Function Groups: Graphics, Keyboard

Related to: FocusID

Format: NextFocusID(Object, ID)

Parameters:

Object

Required. Any expression for the object that defines the window where the focus is found.

ID

Required. Any numeric expression for the new focus ID number.

Comments: This statement is useful for forcing the focus to a specific object in a window.

Example:

```
If Change(myVar, 0);
[
  IfThen(FocusID(Self()) != myFocus,
       NextFocusID(Self(), myFocus));
]
```

These statements check a variable called myVar, whose value is set in previous code by an edit field, and whenever its value changes (i.e. the edit field received the focus) the focus is checked, then reverted to another object (such as an OK button).

Normal

Note: Deprecated. Do not use in new code.

(Alarm Manager module)

Description: Use this function to tell the Alarm Manager when an alarm clears. This subroutine will deactivate the alarm. It will not affect the unacknowledged status.

Returns: Numeric

Usage: Script Only.

Function Groups: Alarm

Related to: Register (Alarm Manager) | CurrentTime | IsActive

Format: \AlarmManager\Normal(AlarmObject[, EventTime]);

Parameters:

AlarmObject

Required. The object value for the alarm that was passed to the Register subroutine.

EventTime

Optional. The time stamp to use when adding this event to the alarm lists. If invalid or not defined, the default is CurrentTime().

Comments: The Normal subroutine always returns "0".

Example:

To avoid an IF 1 condition when normalizing an alarm, it is common practice to include a variable to ensure that the script runs only once. This should take its value from the current alarm state.

```
Init [
  AlarmOn  =  AlarmManager\IsActive(MyAlarm);
]
Main [
IF value < SomeSetPoint && AlarmOn;
[
  AlarmOn = 0;
  AlarmManager\Active(MyAlarmObj);
]
```

Normalize

Description: Returns a normalized value.

Returns: Nothing

Usage: Steady State only. See: Rules for Usage.

Function Groups: Graphics

Related to: Limit | Scale | Tag

Format: Normalize(Value, LowScale, HighScale)

Parameters:

Value

Required. Any numeric expression which represents the value to normalize.

LowScale

Required. Any numeric expression, which represents the lowest normal scaled value of Value. This is not a limit.

HighScale

Required. Any numeric expression, which represents the highest normal scaled value of Value. This is not a limit.

Comments:

This function encapsulates an expression with low and high scale values. Typically, this is used in a trajectory, rotation or layered graphics function for scaling. The return value is a Normalize value. The function does not limit the value to be within the range defined by LowScale and HighScale, but rather, makes it such that when Value equals HighScale the object (if using in a layered graphics function) will fill its bounding box. If Value exceeds HighScale, the object will extend past its bounding box.

Low and high scale values may be calculated expressions. The compiler will reduce them to constants if they evaluate to constants. If any parameters are invalid, the resulting value will still be valid normalized value.

For example:

```
Valid(Normalize(Invalid, 1, 2))
```

will evaluate to 1 (true) and

```
ValueType(Valid(Normalize(Invalid, 1, 2)))
```

will evaluate to 21 - a normalize value.

Example:

```
GUIRectangle(0, 100, 100, 0 { Bounding box of rectangle },
             1, 1, 1 { No scaling of left, bottom or right },
             Normalize(reactorTemp, 0, 150), 1
             { Scale top only, not whole object },
             0, 0 { No trajectory or rotation },
             1, 0 { Rectangle is visible; reserved },
             0, 0, 0 { Cannot be focused },
             12, 15 { Bright red outlined in white });
```

The variable reactorTemp will be scaled for upper and lower values of 0 and 150. As the value of reactorTemp changes, the top of the rectangle will move proportionately. If it exceeds 150, the top of the rectangle will move outside of the original bounding box.

NormalTrip

Deprecated. Do not use in new code. (Alarm Manager module)

Description: This subroutine will deactivate an alarm and signal it as unacknowledged.

Returns: Numeric

Usage: Script Only.

Function Groups: Alarm

Related to: Register (Alarm Manager) | CurrentTime

Format: \AlarmManager\NormalTrip(AlarmObject[, EventTime]);

Parameters:

AlarmObject

Required. The object value for the alarm that was passed to the Register subroutine.

EventTime

Optional. The time stamp to use when adding this event to the alarm lists. If invalid or not defined, the default is CurrentTime().

Comments: The NormalTrip subroutine always returns "0".

Not

Description: Returns the result of a 32 bit unsigned bitwise logical NOT operation.

Returns: 32 bit unsigned integer

Function Groups: Bitwise Operation

Usage: Script or steady state.

Related to: And | Or | XOr

Format: Not(Value)

Parameters:

Value

Required. Any numeric expression. The expression will be truncated to a 32 bit unsigned integer.

Comments: If Value is invalid, the return value is invalid.

Examples:

```
r = Not(1);
s = Not(-1);
t = Not(0);
u = Not(-3);
```

The values of r, s, t and u will be -2 (0xFFFFFFFE), 0 (0x00000000), -1 (0xFFFFFFFF) and 2 (0x00000002) respectively.

NotifyVIC

Description: Sends a message to the VTScada Internet Client (VIC). The message sent depends on the parameter given to the function.

Warning: For use by advanced programmers only. Effective use of this function requires a thorough understanding of VTScada programming.

Returns: Nothing

Usage: Script Only.

Function Groups: VTScada Internet Client

Related to:

Format: NotifyVIC(Value)

Parameters:

Value

Required. Any numeric expression from the following table.

Value	Meaning
0	No Op
1	App is stopping
2	Authentication Failure
3	User stopped the session
4	User logged out
5	The operator terminated the session (from the Internet Client Monitor)
6	The operator forced a server changeover (from the Internet Client Monitor)
7	Refused due to license limits

Comments: none

Now

Description: Returns the current time in seconds since midnight.

Returns: Numeric

Usage: Steady State only. See: Rules for Usage.

Function Groups: Time and Date

Related to: Seconds | Time

Format: Now(Interval)

Parameters:

Interval

Required. Any numeric expression giving the update time in seconds. Fractions of a second may be specified however, Interval must be greater than or equal to 0.

Comments: This function is similar to Seconds, except that the update interval can be specified. The Now function is re-evaluated every Interval seconds, and the return value is the time in seconds since midnight, rounded to the next lowest whole multiple of Interval.

Now will report time to an accuracy of 0.001 seconds, however it would be unreasonable to expect the function to trigger reliably each millisecond under normal operating conditions. Your CPU speed and the load placed upon it by other processes will both affect the maximum number of times that Now can be triggered each second.

Example:

```
ZText(20, 20 { Window location of text },
      Time( { Convert from seconds to standard time }
            Now(1 { Update the time every second }),
            2 { Use hh:mm:ss format }),
      5, 0 { Color is dark magenta, use default font });
```

This displays the time on the screen. It updates every second on the second.

Related Information:

See: "Latching and Resetting Functions" in the VTScada Programmer's Guide

NParm

Description: Returns the number of parameters listed in a module instance.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Compilation and On-Line Modifications, Advanced Module

Related to: AddParameter | FormalParms | NumParms | Parameter | RemoveParameter | ResetParm

Format: NParm(Object)

Parameters:

Object

Required. Any object value, variable, or expression for a Module.

Comments: This function is for experienced users, and is not needed for normal operation. Any user-defined module can be called with any number of parameters. This function returns the actual number of parameters in the call made to the module instance Object. For launched modules the return value is the minimum of the actual and the formal parameters.

Example:

```
<
Show
(
  parm1;
  parm2;
)
Main [
  myParms = NParm(Self());
]
>
```

When called in steady-state as Show("A", "b", "C"), the value of myParms will be "3". When launched as Launch("Show", Invalid, Invalid, "A", "b", "C"), the value of myParms will be "2".

NumericParameterEdit

Description: Wrapper for ParameterEdit, used when adding a numeric parameter to a control.

Returns: Self

Usage: Steady State only.

Function Groups: Basic Module, Variable

Related to: ParameterEdit

Format: NumericParameterEdit(Left, Bottom, Right, Top, ParmVal, ParmCodePtr, TagObjPreferred, Title, PtrWaitClose, DialogRoot, MinLimit, MaxLimit, PTypeIdx[, TitleWidth, ShowDrawnTagProperty, FocusID, Type])

Parameters:

Left

Required. Any numeric expression for the left edge of the object.

Bottom

Required. Any numeric expression for the bottom edge of the object.

Right

Required. Any numeric expression for the right edge of the object.

Top

Required. Any numeric expression for the top edge of the object.

ParmVal

The parameter value to be altered.

ParmCodePtr

The code pointer to the parameter.

TagObjPreferred

Set to TRUE in order to obtain a tag object when possible.

Title

The title to display.

PtrWaitClose

Wait to close.

DialogRoot

Root dialog calling this control.

MinLimit

Minimum limit of the selection.

MaxLimit

Maximum limit of the selection.

PTypeIdx

Index of the Parameter type selection.

TitleWidth

Optional, number of pixels allotted for the width of the title.

ShowDrawnTagProperty

Optional Boolean, set to TRUE to show "Drawn Tag Property". Defaults to TRUE.

FocusID

Optional value for the focus ID of this control.

Type

Optional value specifying the type for the parameter. Defaults to Double.

Comments: Automatically determines what module to use for display and entry of the existing value.

Example:

```
ParmEditObj = \NumericParameterEdit(0, 2 * (EditHt + Space), PanelWd,
0,
                          Parms[#DataSource]       { Parameter Value
```

```
},                              ParmCodePtrs[#DataSource] { Pointer to Parm
Code },
                                1                         { Tag object pre-
ferred },
                                \DataSourceLabel          { Title
},
                                &SubWaitClose             { Wait to close
},
                                DialogRoot                { Calling dialog
window},
                                LowScale                  { MinLimit
},
                                HighScale                 { MaxLimit
});
```

NumInstances

Description: Returns the number of module instances currently running.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Basic Module

Related to: GetInstance | Instance | CalledInstances

Format: NumInstances(Module)

Parameters:

Module

Required. Any module or object value of the module to count.

Comments: This function is useful for counting both the number of active instances of the current module as well as the number of active instances of another module.

Example:

```
numThis = NumInstances(Self());
If ! Valid(numAnother);
[
  numAnother = NumInstances(FindVariable("Calculations", Self(),
  0, 1));
]
```

These statements count the number of active instances of the current module and the Calculations module. Notice that the script is necessary in the second case, not because of the NumInstances function, but because of the FindVariable function.

NumParms

Description: Returns the number of parameters of a statement.

Warning: This function should be used by advanced users only.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Compilation and On-Line Modifications, Advanced Module

Related to: FormalParms | NParm

Format: NumParms(Statement)

Parameters:

Statement

Required. Any expression for the statement code value or code pointer value.

Comments: This function returns the number of parameters of a statement.

NumSelected

Description: Returns the number of selected graphics statements in a window.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Graphics

Related to: SelectGraphic | UnselectGraphics | UnselectObject

Format: NumSelected(Object)

Parameters:

Object

Required. Any expression for the object that defines the window.

Example:

```
selGraphic = NumSelected(CurrentWindow());
```

This will set selGraphic to the number of currently selected objects in the window that the mouse is presently over.

NumSets

Description: Returns the number of statements that are currently active in setting a particular variable.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Compilation and On-Line Modifications, Variable

Related to: Watch

Format: NumSets(Variable)

Parameters:

Variable

Required. Any variable for which the number of sets is required.

Example:

```
setsOnX = NumSets(x);
```

This will set setsOnX to the number of statements that are currently active in setting the value of variable x.

NumVariables

Description: Returns the number of variables in a module.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Compilation and On-Line Modifications, Variable

Related to: SelectGraphic | UnselectGraphics | UnselectObject

Format: NumVariables(Module)

Parameters:

Module

Required. Any expression for the object or module value.

Example:

```
numVars = NumVariables(Self());
```

This will set numVars to the number of variables in the current module.

O Functions

The sections that follow identify all VTScada functions beginning with "O".

ODBC

Description Performs an ODBC command and returns a dynamically allocated array if required.

Returns Array

Usage Script Only.

Function Groups Database and Data Source, ODBC

Related to: ODBCConfigureData | ODBCConnect | ODBCDisconnect | ODBCSources | ODBCStatus | ODBCTables | TODBC |

TODBCConnect | TODBCDisconnect

Format: ODBC(DB, SQLCommand [, Attrib, ErrorMsg, SQLState, ErrorCode])

Note: Refer to the comments section of TODBC for a discussion on the differences between blocking and non-blocking ODBC calls.

Parameters

DB

Required. An ODBC value for the ODBC database as returned by ODBCConnect.

SQLCommand

Required. Any text expression for the SQL command to perform on the ODBC database driver. If the query involves long binary data types, then a structure should be used. See examples.

Attrib

An optional parameter that returns a 2D array, with each row in the array holding detailed attributes for one column of the result set, as per the following

Attrib	Attribute
Attrib[col][0]	Name of the column
Attrib[col][1]	Type of data VTScada will return for the column: 0 == text 1 == numeric
Attrib[col][2]	Type SQL Data Types indication for the field:
Attrib[col][3]	A numeric value that is either the maximum or actual character length of a character string or binary data type. It is the maximum character length for a fixed-length data type, or the actual character length for a variable-length data type.

ErrorMsg:

A descriptive error or status message, returned by the function. If valid, both ErrorMsg and SQLState will be valid.

SQLState:

A 5-character SQL State code. SQL State codes are defined by Microsoft and by the vendor of each ODBC driver.

ErrorCode:

An unsuccessful operation always returns a non-zero value, which is a numeric error code specific to the

DBMS vendor's ODBC driver or Microsoft's ODBC Driver Manager.

A successful operation will always return a 0. ErrorMsg and SQLState may or may not be set valid in the event of a successful connection. If set valid, they should be examined for relevant status information.

Comments:

There may or may not be a return value for this function, depending on the nature of the SQL command that was executed. If a return value exists, it will be a dynamically allocated, two-dimensional array that contains the rows resulting from the query. The format for the array is Result [Field][Record].

If any error, no matter how minor, occurs as a result of the SQL command, and if the ODBCConnect that connected to the database had its Disconnect parameter set true, then the value of DB will become invalid (i.e. the connection to the database will be dropped).

All ODBC operations can result in one or more status or error conditions arising. VTScada records the entire set of status/error conditions arising and buffers them internally. Use the ODBCStatus function to retrieve the entire set. Only the first condition that occurred or, if both error and advisory conditions occur, the first error condition, is returned in the ErrorMsg and SQLState values.

From VTS 10.0 onwards, VTScada uses ODBC 3.x compliant operations (formerly ODBC 2.x). This has the side effect that different SQLState return values are returned for some SQLState values. If you have written code that depends on the value returned by SQLState, you may need to change the value you expect. See http://msdn.microsoft.com/en-us/library/ms712451%28VS.85%29.aspx for a reference on the value changes.

In the case of the optional parameters, any parameter that is not required may be set to 0 if it is followed by a valid parameter, or may be simply omitted if no valid parameters

follow it.

This command requires a knowledge of SQL (Structured Query Language). The examples provide several SQL statements which you can use as templates.

Example 1

Optional entities are enclosed in square brackets [ ], while required ones are enclosed by angled brackets < >. Italicized text represents names of tables, fields, etc. Embedded quotes need to appear twice to signify to VTScada that they are part of the string, and do mark the close of the string.

To create a table, the basic format is:

```
Create Table "TableName" (< list of fields/types >)
```

where

< list of fields/types > is a comma separated list of fields and their types (the field and type are separated by a white space character) that define the table. The field names must be enclosed in quotation marks if they duplicate an SQL reserved word or if they contain a white space character. Field types include (but are not limited to):

Int or Integer, SmallInt, Float, Real, Double, Precision, Dec(p,d) or Decimal(p,d), Numeric(p, d)

Char(n) or Character(n), VarChar(n) or Char Varying(n) or Character Varying(n)

Bit(n), Bit Varying(n)

Date, Time

where p is the precision (total number of decimal digits) and d is the number of places after the decimal point.

To create a table for a custom tag type called Motor, the SQL command might look something like the following:

```
Create Table "Motor" ("Name" Char(32), "Area" Char(32), "Description"
Char(32), "Input" Char(32), "Status" Int, "Temperature" Decimal(5,
1))
```

To insert an entry into a table, the basic format is:

```
Insert Into "TableName"
[(<list of fields >)]
Values (<list of values >)
```

where
[(< list of fields >)] is an optional clause that is a comma separated list of fields defining which fields to assign the values to. The field names must be enclosed in quotation marks if they duplicate an SQL reserved word or if they contain a white space character. If this field list is omitted, all fields must have an assigned value, even if they are null, in which case the reserved word NULL (no quotes) is used.
< list of values > is a comma separated list of field values that define the record and take the form:

```
'Value'
```

Values must be enclosed in single quotes if they are text strings.

Example 2:

To insert a record with binary large objects:

```
{ Declare and initialize the structure }
ODBCQuery STRUCT [
  QueryString;
  Parameters;
];
Query = ODBCQuery();

Query\QueryString = "INSERT INTO TestLargeBlobs (id, Blob1, Blob2)
VALUES (1, ?, ?)";
Query\Parameters = New(2);
Query\Parameters[0] = MakeBuff(10000000, 65);
Query\Parameters[1] = MakeBuff(20000000, 66);
ODBC(DBHandle, Query);
```

Similarly, to do an update:

```
Query = ODBCQuery();
Query\QueryString = "UPDATE TestLargeBlobs SET Blob2=? WHERE id=1";
Query\Parameters = MakeBuff(30000000, 67);
ODBC(DBHandle, Query);
```

Example 3:

To insert a record with a valid name and status (but all other fields invalid) into the table created in the previous example, the SQL command might look something like the following:

```
Insert Into "Motor" ("Name", "Status") Values ('Motor 1234', 1)
```

To retrieve data from a table, the basic format is:

```
Select [ number of records ]< list of fields >
From <list of tables >
[Where < conditions >]
[Order By < list of fields >]
```

where
[number of records] is an optional statement that limits the number of records retrieved. It takes the form:

```
Top N
```

where N is the number of records.
< list of fields > is a comma separated list of fields to retrieve for each record, or by which the records are sorted. The field names must be enclosed in quotation marks if they duplicate an SQL reserved word or if they contain a white space character. If all of the fields for a record are to be retrieved, an asterisk should be used. The asterisk must not be enclosed in quotation marks.
<list of tables > is a comma separated list of tables from which to retrieve the data. As with the attribute list, the table names must be enclosed in quotation marks if they duplicate an SQL reserved word or if they contain a white space character.
[< conditions >] is an optional clause giving the list of conditions that take the form:

```
"FieldName" = 'Value'
```

where FieldName is the actual name of the field and Value is the numeric or text value to match. Note that once again, each field name need only be enclosed in quotation marks if it duplicates an SQL reserved word or contains a white space character. Similarly, the value needs to be enclosed in single quotes only if it is a text string. Multiple conditions are separated by the key word And.

Example 4

Suppose that in a VTScada application a user wanted retrieve the first 10 entries of an alphabetical list of names and descriptions for standard analog input tags that belonged to the system area and had questionable data:

```
Select Top 10 "Name", "Description" From "AnalogInput"
   Where "Area" = 'System' And "Questionable = 1 Order by "Name"
```

To modify an entry in a table, the basic format is:

```
Update "TableName" Set <list of fields/values >
```

Where < conditions >

where

< list of fields/values > is a comma separated list of fields and their values that define the record and take the form:

```
"FieldName" = 'Value'
```

Field names must be enclosed in quotation marks if they duplicate an SQL reserved word or if they contain a white space character. Values must be enclosed in single quotes if they are text strings.

< conditions > is a list of conditions that define which record to modify. For more information, see the < conditions > section in item number 3.

Example 5

To change an existing record (tag) in the standard analog input table (tag type), the SQL command might look something like the following:

```
Update "AnalogInput" Set "Area" = 'System', "UnscaledMin" = 10,
"UnscaledMax" = 80 Where "Name" = 'AI36'
```

To delete an entry from a table, the basic format is:

```
Delete From "TableName" Where < conditions >
```

where

< conditions > is a list of conditions that define which record to delete. For more information, see the < conditions > section in item number 2.

Example 6

To delete the record that was modified in the previous example entirely, the SQL command might look something like the following:

```
Delete From "AnalogInput" Where "Name" = 'AI36'
```

Example 7

To connect to an ODBC data source and create a table whose name is held in the variable tableName, and whose text string lengths are limited to the value of the variable maxLen, the calls might look something like the following:

```
If Valid(dsName) Main;
[
  dbHandle = ODBCConnect(dsName, "", "", eMsg, eState, eCode);
  eType = ODBCStatus(0);
  IfThen(eType != 0,
        Slay(Self(), 0);
  );
  commandString = Concat("Create Table """, tableName,
      """ (""Name"" Char(", maxLen, "), ""Area"" Char(", maxLen,
      ""Description""", Char(", maxLen, "), "Input" Char(", maxLen,
      "), "Status" Int, "Temperature" Decimal(5, 1));
  result = ODBC(dbHandle, commandString, fieldAttrib, eMsg,
               eState, eCode);
  IfThen(eType != 0,
   Slay(Self(), 0);
  );
]
```

The following example shows an SQL command placed into a VTScada parameter:

```
ODBC(DB, "Create Table ""Motor"" (""Name"" Char(32), ""Area""  Char
(32), ""Description"" Char(32), ""Input"" Char(32), ""Status"" Int,
""Temperature"" Decimal(5, 1))");
```

Note that all embedded quotes need to appear twice to tell VTScada that they are part of the string and not the end of the string. This applies to all text constants used anywhere in VTScada.

ODBCBeginTrans

Description:	Indicates to a specified ODBC-compliant database that a transaction is to be started.
Returns:	Nothing (return values in parameters)

Usage: Script Only.

Function Groups: Database and Data Source, ODBC

Related to: ODBCCommit | ODBCRollback | TODBCBeginTrans | TODBCCommit | TODBCRollback

Format: ODBCBeginTrans(DB [, ErrorMsg, SQLState, ErrorCode])

Parameters:

DB

Required. An ODBC-compliant database as returned by ODBCConnect.

ErrorMsg

A descriptive error or status message, returned by the function. If valid, both ErrorMsg and SQLState will be valid.

SQLState

A 5-character SQL State code. SQL State codes are defined by Microsoft and by the vendor of each ODBC driver.

ErrorCode

An unsuccessful operation always returns a non-zero value, which is a numeric error code specific to the DBMS vendor's ODBC driver or Microsoft's ODBC Driver Manager.

A successful operation will always return a 0. ErrorMsg and SQLState may or may not be set valid in the event of a successful connection. If set valid, they should be examined for relevant status information.

Comments: A transaction is a unit of work that is done as a single operation. The operation succeeds or fails as a whole. ODBCBeginTrans indicates that a transaction is to be started on the specified ODBC database. Note that each ODBC database driver may have different levels of transaction support (or

none at all), and the documentation for that driver should be consulted to determine the level of transaction support. The transaction should be terminated either with an ODBCCommit or an ODBCRollback.
If any error (no matter how minor) occurs as a result of the statement, and the TODBCConnect or ODBCConnect that connected to the database had its Disconnect parameter set to true, the value of DB will become invalid (i.e. the connection to the database will be dropped).

Example:

```
[
  ODBCBeginTrans(DB);
  ODBC(DB, "DELETE * FROM TEST");
  ODBC(DB, "INSERT INTO TEST VALUES('KeyString', 'Value')");
  ODBCCommit(DB);
]
```

ODBCCommit

Description: Indicates to a specified ODBC-compliant database that a transaction is to be committed.

Returns: Nothing (return values in parameters)

Usage: Script Only.

Function Groups: Database and Data Source, ODBC

Related to: ODBCBeginTrans | ODBCRollback | TODBCBeginTrans | TODBCCommit | TODBCRollback

Format: ODBCCommit(DB [, ErrorMsg, SQLState, ErrorCode])

Parameters:

DB

Required. An ODBC value for the specified ODBC database as returned by ODBCConnect.

ErrorMsg

A descriptive error or status message, returned by the

function. If valid, both ErrorMsg and SQLState will be valid.

SQLState

A 5-character SQL State code. SQL State codes are defined by Microsoft and by the vendor of each ODBC driver.

ErrorCode

An unsuccessful operation always returns a non-zero value, which is a numeric error code specific to the DBMS vendor's ODBC driver or Microsoft's ODBC Driver Manager.
A successful operation will always return a 0. ErrorMsg and SQLState may or may not be set valid in the event of a successful connection. If set valid, they should be examined for relevant status information.

Comments: Commits a transaction defined as all the SQL statements executed on an ODBC-compliant database since the transaction began.
If any error, no matter how minor, occurs as a result of the statement, and the TODBCConnect or ODBCConnect that connected to the database had its Disconnect parameter set to true, the value of DB will become invalid (i.e. the connection to the database will be dropped).

Example:

```
[
  ODBCBeginTrans(DB);
  ODBC(DB, "DELETE * FROM TEST");
  ODBC(DB, "INSERT INTO TEST VALUES('KeyString', 'Value')");
  ODBCCommit(DB);
]
```

ODBCConfigureData

Description: Configures an ODBC data source and returns its error code.

Returns: Numeric

Usage: Script Only.

Function Groups: Database and Data Source, ODBC

Related to: ODBC | ODBCConnect | ODBCDisconnect | ODBCSources | ODBCStatus | ODBCTables | TODBC | TODBCConnect | TODBCDisconnect

Format: ODBCConfigureData(Mode, DriverName [, Settings])

Parameters:

Mode

Required. Any numeric expression for the mode as follows:

Mode	Meaning
0	Add data source
1	Configure data source
2	Remove data source

Note: In order for 64-bit VTScada to work with 64-bit data sources, add 64 to the Mode parameter. 32-bit VTScada cannot configure a 64-bit data source.

DriverName

Required. Any text expression for the ODBC driver name, as configured in the ODBC setup menu under Microsoft Windows™. In the case of Excel, use "Microsoft Excel Driver (*.xls)".

Settings

A variety of optional parameters that are text expressions, giving the variable and value pairs used to specify the configuration of the ODBC data source.
Any number of these parameter pairs may be used by simply listing the text string containing the variable name, followed by the value that the variable is to be set to. The variables to use include

Settings	Description
DBQ	Name of the workbook (database) file
DefaultDir	Workbook directory
Description	Text description for data source
Driver	Path to the driver .DLL
DriverID	Integer ID for the driver
DSN	Data source name
FileType	File type
FirstRowHasNames	Sets if first row contains column names
MaxScanRows	Rows to scan in setting data type of column (range is 0 - 16)
ReadOnly	Sets the database file as read only

If the DriverID option was selected, the Excel driver values that may be used are as follows:

Value	Driver
534	Microsoft Excel 3.0
278	Microsoft Excel 4.0
22	Microsoft Excel 5.0/7.0
790	Microsoft Excel 97

Comments:

If the data source already exists, it will be reconfigured as per the specs given and "0" (no error) will be returned.
The data source name may not contain apostrophes or an ODBC error will occur. The function will return the error code.
Some types of ODBC data sources, such as Microsoft™ Excel, do not require the file to be created prior to executing SQL commands on the data source, but will create a blank file when the first SQL Create Table command is executed. Others, such as Microsoft™ Access require the file to be created prior to execution of any SQL statements.

Note: Please note that configuration of ODBC Data Sources requires write permission to the registry key "HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBC.INI". If the current user doesn't have write permission to the key, then the string "Insufficient permissions" will be returned. If running in Windows Vista, this means that VTScada must be run as Administrator for the function to work.

The following table identifies some possible errors and their meaning:

Error Message	Significance
General Installer Error	An error occurred for which there was no specific ODBC installer error.
Invalid Type of Request	The Mode parm was something other than 0, 1, or 2.
Invalid Driver or Translator Name	The DriverName parm was not a valid ODBC driver name (found in the registry).
Invalid Keyword-value pairs	One of the Settings parms contained a syntax error. For example, spaces are not permitted around the equals sign in the keyword-value

Examples:

```
{ Create a data source }
If ZButton(10, 30, 110, 10, "Create DS", 1);
[
  { Make sure name doesn't contain apostrophes }
  Replace(DSName, 0, StrLen(DSName), "'", "");
  ODBCConfigureData(0, "Microsoft Access Driver (*.mdb)",
                   "DSN", DSName,
                   "DefaultDir", Concat(DSDrive, DSPath),
                   "DBQ", Concat(DSFileName, ".MDB"),
                   "Description", DSDesc,
                   "FileType", "Access",
                   "ReadOnly", 0);
]
{ Create the data source's file }
If ZButton(10, 70, 110, 40, "Create file", 2);
[
  ODBCConfigureData(1, "Microsoft Access Driver (*.mdb)",
                   "DSN", DSName,
                   "CREATE_DB",
                   Concat(DSDrive, DSPath,
                    DSFileName, ".MDB"));
]
{ Delete the data source }
If ZButton(10, 110, 110, 80, "Delete DS", 3);
[
  ODBCConfigureData(2, "Microsoft Access Driver (*.mdb)",
                   "DSN", DSName);
]
{ Create a MS SQL Server data source }
If ZButton(10, 150, 110, 120, "Create SQL DS", 3);
[
  ErrorCode = ODBCConfigureData(0, "SQL Server", "DSN",ODBCName,
                                "Description",
                                 "SQL server connection created by
VTS.",
                                 "SERVER",
                                 ComputerName, "NETWORK",
                                "DBMSSOCN" );
]
```

To remove a 64-bit data source (using 64-bit VTS):

```
ODBCConfigureData(66, …)
```

To add a 32-bit data source (using either 64-bit or 32-bit VTS):

```
ODBCConfigureData(0, …)
```

ODBCConnect

Description: Forms a connection to an ODBC-compliant database and returns the ODBC value associated with that database.

Returns: ODBC database

Usage: Script Only.

Function Groups: Database and Data Source, ODBC

Related to: ODBC | ODBCConfigureData | ODBCDisconnect | ODBCSources | ODBCStatus | ODBCTables | TODBC | TODBCConnect | TODBCDisconnect

Format: ODBCConnect(DSName, UserName, Password [, ErrorMsg, SQLState, ErrorCode, Disconnect, LoginTimeout, ConnectTimeout])

Parameters:

DSName

Required. Any text expression for the ODBC data source name, as configured in the ODBC setup menu under Microsoft Windows™.

UserName

Required. Any text expression for the ODBC login user name.

Password

Required. Any text expression for the ODBC login password.

ErrorMsg

A descriptive error or status message, returned by the function. If valid, both ErrorMsg and SQLState will be valid.

SQLState

A 5-character SQL State code. SQL State codes are defined by Microsoft and by the vendor of each ODBC driver.

ErrorCode

An unsuccessful operation always returns a non-zero

value, which is a numeric error code specific to the DBMS vendor's ODBC driver or Microsoft's ODBC Driver Manager.
A successful operation will always return a 0. ErrorMsg and SQLState may or may not be set valid in the event of a successful connection. If set valid, they should be examined for relevant status information.

Disconnect

An optional parameter that is any logical expression that determines how errors are to be handled. If true (non-0), the connection to the database will be disconnected should any error (no matter how minor) occur.
If false (0) an error will not cause a disconnect to occur. The default value is false.

LoginTimeout

Optional. An SQLUINTEGER (unsigned long) value corresponding to the number of seconds to wait for a login request to complete before returning to the application. The default is driver-dependent. If the value is 0, then the timeout is disabled and a connection attempt will wait indefinitely.
If the specified timeout exceeds the maximum login timeout in the data source, then the driver substitutes that value and returns SQLSTATE 01S02 (Option value changed).

ConnectTimeout

Optional. An SQLUINTEGER value corresponding to the number of seconds to wait for any request on the connection to complete before returning to the application. The driver should return SQLSTATE HYT00 (Timeout expired) any time that it is possible to time out in a situation not associated with query execution or login.

If the value is equal to 0 (the default) then there is no timeout.

Comments: If the UserName and Password parameters are specified as invalid, the DSName parameter is treated as a literal connection string for the ODBC connection. For example, in the statement ODBCConnect("DSN=MyData;UID=;PWD=", Invalid, Invalid), the first parameter is used to set the attributes of the connection, which makes this statement equivalent to ODBCConnect("MyData", "", "").

In the case of the optional parameters, any parameter that is not required may be set to 0 if it is followed by a valid parameter, or may be simply omitted if no valid parameters follow it.

On successful completion of the ODBC and ODBCConnect functions, the native error code will be set to 0, allowing the user to tell if a command that has no result set has been completed. The user should not assume that since a 0 is returned, then the command has been executed successfully. Some drivers (such as Excel) will return a 0 in the native error code, even when an error has occurred.

All ODBC operations can result in one or more status or error conditions arising. VTScada records the entire set of status/error conditions arising and buffers them internally. Use the ODBCStatus function to retrieve the entire set. Only the first condition that occurred or, if both error and advisory conditions occur, the first error condition, is returned in the

ErrorMsg and SQLState values.

From VTS 10.0 onwards, VTS uses ODBC 3.x compliant operations (formerly ODBC 2.x). This has the side effect that different SQLState return values are returned for some SQLState values. If you have written code that depends on the value returned by SQLState, you may need to change the value you expect. See http://msdn.microsoft.com/en-us/library/ms712451%28VS.85%29.aspx for a reference on the value changes.

64-bit VTScada is able to connect to either 64-bit or 32-bit ODBC data sources. ODBCConnect will first try to connect to the database through a 64-bit ODBC driver. If this fails for any reason it will then try the connection through a 32-bit ODBC driver. This means that any ODBC code that worked under 32-bit VTScada should not need to be modified for use with 64-bit VTScada, but 64-bit VTScada has the extra ability of being able to use 64-bit ODBC drivers.

ODBCDisconnect

Description: Stops a connection to the ODBC database.

Returns: Nothing

Usage: Script Only.

Function Groups: Database and Data Source, ODBC

Related to: ODBC | ODBCConfigureData | ODBCConnect | ODBCSources | ODBCStatus | ODBCTables | TODBC | TODBCConnect | TODBCDisconnect

Format: ODBCDisconnect(DB)

Parameters:

DB

Required. An ODBC value for the ODBC database as returned by ODBCConnect.

Comments: none.

ODBCRollback

Description: Indicates to a specified ODBC-compliant database that a transaction is to be rolled back (discarded).

Returns: Nothing (return values in parameters)

Usage: Script Only.

Function Groups: Database and Data Source, ODBC

Related to: ODBCBeginTrans | ODBCCommit | TODBCBeginTrans | TODBCCommit | TODBCRollback

Format: ODBCRollback(DB [, ErrorMsg, SQLState, ErrorCode])

Parameters:

DB

Required. An ODBC value for the specified ODBC database as returned by ODBCConnect.

ErrorMsg

A descriptive error or status message, returned by the function. If valid, both ErrorMsg and SQLState will be valid.

SQLState

A 5-character SQL State code. SQL State codes are defined by Microsoft and by the vendor of each ODBC driver.

ErrorCode

An unsuccessful operation always returns a non-zero value, which is a numeric error code specific to the

DBMS vendor's ODBC driver or Microsoft's ODBC Driver Manager.
A successful operation will always return a 0. ErrorMsg and SQLState may or may not be set valid in the event of a successful connection. If set valid, they should be examined for relevant status information.

Comments: Discards a transaction defined as all the SQL statements executed on an ODBC-compliant database since the transaction began.
If any error, no matter how minor, occurs as a result of the statement, and the TODBCConnect or ODBCConnect that connected to the database had its Disconnect parameter set to true, the value of DB will become invalid (i.e. the connection to the database will be dropped).

Example:

```
[
  ODBCBeginTrans(DB);
  ODBC(DB, "DELETE * FROM TEST");
  ODBC(DB, "INSERT INTO TEST VALUES('KeyString', 'Value')");
  ODBCRollback(DB);
]
```

ODBCSources

Description: Retrieves a list of ODBC data sources and returns it as a dynamically allocated array.

Returns: Array

Usage: Script Only.

Function Groups: Database and Data Source, ODBC

Related to: ODBC | ODBCConfigureData | ODBCConnect | ODBCDisconnect | ODBCStatus | ODBCTables | TODBC | TODBCConnect | TODBCDisconnect

Format: ODBCSources()

Parameters: None

Comments: The return value is a pointer to a two-dimensional array, which contains a list of ODBC data sources. The first row of the array ([0][N]) contains the data source names, while the second describes the driver.
64-bit VTScada only… The returned list will include both 32-bit and 64-bit data sources.

Example:

```
If ! Valid(sources);
[
  sources = ODBCSources() { Find the list of sources };
  count = ArraySize(sources, 1) { The number of sources };
]
```

This will obtain the list and number of ODBC data sources.

ODBCStatus

Description: Returns the requested information about the last ODBC statement to execute.

Returns: Varies – see comments

Usage: Script Only.

Function Groups: Database and Data Source, ODBC

Related to: ODBC | ODBCConfigureData | ODBCConnect | ODBCDisconnect | ODBCSources | ODBCTables | TODBC | TODBCConnect | TODBCDisconnect

Format: ODBCStatus(Option[, ODBCHandle])

Parameters:

Option

Required. Any numeric expression for the option to indicate what information is desired about the last ODBC statement executed as follows:

Option	Information desired
0	Return 0 if information or 1 if error
1	SQL state text
2	Native error code
3	ODBC error text
4	Error text length

ODBCHandle

Optional ODBC Handle value, as returned from ODBCConnect or TODBCConnect, for which you want a status report.
Omitting the ODBCHandle parameter returns a status report for the last ODBC operation that was executed. As ODBC operations can be concurrently executed, there is no guarantee that this is the operation for which you intend to obtain status, unless you explicitly provide an ODBC handle value

Comments: The return value for this function may be a single value, an array of values (if more than one error was generated), or invalid (if the last ODBC operation did not generate any status information).
This function may be used with threaded ODBC calls, however, if multiple ODBC calls are executing simultaneously, there is no indication as to which one generated the message.

ODBCTables

Description: Retrieves a list of the tables present in an ODBC-compliant database and returns it as a dynamically allocated array.

Returns: Array

Usage: Script Only.

Function Groups: Database and Data Source, ODBC

Related to: ODBC | ODBCConfigureData | ODBCConnect | ODBCDisconnect | ODBCSources | ODBCStatus | TODBC | TODBCConnect | TODBCDisconnect

Format: ODBCTables(DB [, Search][, TableType])

Parameters:

DB

Required. An ODBC value for the ODBC database as returned by ODBCConnect.

Search

An optional parameter which is any text string indicating the pattern to match for table names.
If this parameter is omitted, the search pattern defaults to "%", where the percent sign is the SQL wildcard (i.e. all table names are returned).

TableType

An optional parameter which is a list of table types to match. Parameters include the following:
"TABLE", "VIEW", "SYSTEM TABLE",..., or a data source-specific type name.

Comments: If TableType is not an empty string, it must contain a list of comma-separated values for the types of interest. Each value may be enclosed in single quotation marks (') or unquoted for example, 'TABLE', 'VIEW' or TABLE, VIEW. An application should always specify the table type in uppercase. If the data source does not support a specified table type, no results will be returned for that type

Example:

```
If 1 Next;
[
 tables = ODBCTables(dBase, "Analog%");
]
```

This will obtain the list of all tables in the database beginning with the string "Analog".

OffNormal

Note: Deprecated. Do not use in new code.

(Alarm Manager module)

Description: This subroutine will activate the alarm. However, it will not affect the unacknowledged status.

Returns: Numeric

Function Groups: Alarm

Usage: Script Only.

Related to: Register (Alarm Manager) | CurrentTime

Format: \AlarmManager\OffNormal(AlarmObject[, EventTime]);

Parameters:

AlarmObject

Required. The object value for the alarm that was passed to the Register subroutine.

EventTime

Optional. The time stamp to use when adding this event to the alarm lists. If invalid or not defined, the default is CurrentTime().

Comments: The OffNormal subroutine always returns "0".

Ones

Description: Returns the number of bits set in an integer number.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Bitwise Operation

Related to: Bit | SetBit

Format: Ones(Value)

Parameters:

Value

Required. Any numeric expression.

Comments: Value must be a valid number and is truncated to an integer. The number of bits set is returned.

Examples:

```
a = Ones(0b010110);
b = Ones(0b001100);
c = Ones(2.3);
```

The values of a, b and c will be 3, 2 and 1 respectively.

OpChange

Description: Wrapper for TagMigrator\OpChange. Performs an immediate deploy of a single tag change without disturbing any other tag changes already in place on the local branch. The tags must already exist.

Returns: Object value of the worker module.

Usage: Script Only.

Function Groups: Configuration

Related to: SimpleOpChange | ModifyTags

Format: \Code\OpChange(TagName, ParmsDict, User[, Comment, Merge])

Parameters:

Tagname

Required. The name of the tag to be changed.

ParmsDict

Required. A dictionary of tag parameters to change. Must also include the Name parameter. Refer to the examples in ModifyTags.

User

Required. The user responsible for the operational change.

Comment

Optional text. A descriptive comment about the change.

Merge

Optional Boolean. Defaults to TRUE. Controls whether the parameters are to be merged with the existing tag's parameters. If false, the parameters replace the tag's parameters and are expected to contain user and timestamp metadata.

Comments: If your application is set to auto-deploy, then OpChange and ModifyTags accomplish the same result.
If auto-deploy is not set, then there is a difference in the result of the two functions. ModifyTags makes a tag change locally, whereas OpChange will immediately deploy that tag change. Running a deploy operation after ModifyTags' change, results in the entire tag file (and all the included tags and their parameters) being deployed. OpChange is able to deploy just the one change to the one tag within the file, leaving other modified tags in that file as local changes.

OpChanges are used for any tag change that is expected to be performed during systems operation, as opposed to changes made during configuration. The example is an operator changing a tag's questionable status by using the context menu rather than opening the configuration dialog.

OPCServer

Description: Adds a new top-level branch to the VTScada OPC Server's namespace hierarchy.

Returns: Handle – see comments

Usage: Steady State only.

Function Groups: Network

Related to: SetOPCData

Format: OPCServer(BranchName, CallbackContext)

Parameters:

BranchName

Required. The name of the branch added at the highest level of the hierarchy.

CallbackContext

Required. The object value of a running module containing some or all of the following callback modules (these callback modules must be implemented by the programmer).
In each case the Item ID passed to the callback will *not* contain the top-level branch name specified in the call to OPCServer.

Modules: **OPCGetItemAttributes(itemID)** – Given an OPC item ID, returns a structure containing the attributes of the item. Structure format:

```
ItemAttributes STRUCT [
       AccessRights;
        Type;
       HasChildren
 ];
```

Where:
AccessRights is one of the following values:
1 = READABLE,
2=WRITEABLE,

3= READWRITEABLE

Type is the COM data type of the item value:
2 = VT_I2,
3 = VT_I4,
5 = VT_R8,
8 = VT_BSTR,
11 = VT_BOOL

HasChildren is one of the following values:
0 = has no children,
1 = has children

OPCGetChildNodes(itemID) – Given an OPC item ID, returns an array of strings that are the names of child items.

OPCReadItem(itemID) – Given an OPC item ID, returns a structure containing the value, quality and UTC timestamp of the item. Structure format:

```
OPCVQT Struct [
    Value;
    Quality;
    Timestamp;
  ];
```

OPCWriteItem(itemID, Value) -- Given an OPC item ID and a value, attempts to write the value to the item.

OPCGetInternalName(itemID) – Given an OPC item ID, returns the internal name used for that item. This is useful for the case when two or more different item Ids might refer to the same internal "item".

OPCGetProperties(itemID) – Given an OPC item ID, returns an array of structures describing the properties found on that item. Structure format:

```
  OPCProperty Struct [
    ID           { Property ID number
};
    Type           { Data type of the property
};
    Name         { Name of the property
};
    Description    { Description of the property
};
    InvalidMeansBadQuality  { TRUE to indicate
that an invalid property value implies Bad qual-
ity data                     };
  ];
```

OPCGetPropertyValue(itemID, propertyID) – Given an OPC item ID and a numeric property ID, returns the current value of the specified property.

Comments: This function returns a handle to the OPC server namespace branch (to be used in calls to SetOPCData).

Example:

```
<
{==================== SimulateOPCServer ====================}
{ This module runs a simulated OPC server.                 }
{============================================================}
SimulateOPCServer
(
  ServerName;
)
[
  OPCGetChildNodes          Module;
  OPCGetPropertyValue       Module;
  OPCGetProperties          Module;
  OPCReadItem               Module;
  OPCWriteItem              Module;
  OPCGetInternalName        Module;
  OPCGetItemAttributes Module;

  CONSTANT OPC_QUALITY_BAD = 0x00;
  CONSTANT OPC_QUALITY_UNCERTAIN = 0x40;
  CONSTANT OPC_QUALITY_GOOD = 0xC0;

  CONSTANT VT_EMPTY = 0;
  CONSTANT VT_NULL  = 1;
  CONSTANT VT_I2    = 2;
  CONSTANT VT_I4    = 3;
  CONSTANT VT_R4    = 4;
  CONSTANT VT_R8    = 5;
  CONSTANT VT_CY    = 6;
  CONSTANT VT_DATE  = 7;
  CONSTANT VT_BSTR  = 8;
]
```

```
ServerOn [
  OPCServer(ServerName, Self());
]

<
OPCGetChildNodes
(
  itemID;
)
[
  returnVal = 0;
]

Main [
  If watch(1);
  [
    IfThen(itemID == "",
      returnVal = New(1);
      returnVal[0] = "tags";
    );
    IfThen(itemID == "tags",
      returnVal = New(7);
      returnVal[0] = "ai1";
      returnVal[1] = "ai2";
      returnVal[2] = "NeverSet";
      returnVal[3] = "TextTag";
      returnVal[4] = "ao1";
      returnVal[5] = "AnalogInputs";
      returnVal[6] = "ao2";
    );

    IfThen(itemID == "tags\AnalogInputs",
      returnVal = New(1);
      returnVal[0] = "ai1";
    );
    Return(returnval);
  ]
]
>

<
OPCGetPropertyValue
(
  itemID;
  propertyID;
)
[
  returnVal;
]
Main [
  If watch(1);
  [
    IfThen(propertyID == 5555,
      returnVal = 888;
    );
    IfThen(propertyID == 104,
```

```
      returnVal = 99.5;
    );
    IfThen(propertyID == 5000,
      returnVal = "TestArea";
    );
    Return(returnVal);
  ]
]
>

<
OPCGetProperties
(
  itemID;
)
[
  returnVal;
  I;
  OPCProperty Struct [
    ID                { Property ID number             };
    Type              { Data type of the property      };
    Name              { Name of the property           };
    Description       { Description of the property    };
  ];

  OtherStruct Struct [
    TestComment;
    Name;
    ItemHandles;
    Value;
    Quality;
    Timestamp;
  ];
]
Main [
  If watch(1);
  [
      IfThen(itemID == "tags\ai1",
        returnVal = New(3);
        I = 0;
        returnVal[I++] = OPCProperty(104, VT_R8, "HighRawValue",
  "High Instrument Range");
        returnVal[I++] = OPCProperty(5000, VT_BSTR, "Area", "Area");
        returnVal[I++] = OPCProperty(5555, VT_I2, "Type",
                                     "myVTSProp");
      );
      IfThen(itemID == "tags\ai2",
        returnVal = 0;
      );
    Return(returnVal);
  ]
]
>

<
OPCWriteItem
(
```

```
  itemID;
  Value;
)
Main [
  If watch(1);
  [
   { Do a write to the specified item. }
    Return(0);
  ]
]
>

<
OPCGetInternalName
(
  itemID;
)
[
  retval;
]

Main [
  If watch(1);
  [
    IfThen(itemID == "tags\ai1",
      retval = "ai1";
    );
    IfThen(itemID == "tags\ai2",
      retval = "ai2";
    );
    IfThen(itemID == "tags\TextTag",
      retval = "TextTag";
    );
    IfThen(itemID == "tags\ao1",
      retval = "ao1";
    );
    IfThen(itemID == "tags\ao2",
      retval = "ao2";
    );
    IfThen(itemID == "tags\AnalogInputs\ai1",
      retval = "ai1";
    );

    Return(retval);
  ]
]
>

<
OPCReadItem
(
  itemID;
)
[
  returnVal;

  OPCVQT Struct [
```

```
    Value;
    Quality;
    Timestamp;
  ];

]
Main [
  If watch(1);
  [
    ReadItemTimestamp = CurrentTime();

    IfThen(itemID == "tags\NeverSet",
      returnVal = OPCVQT("NeverSetValue",
                         OPC_QUALITY_UNCERTAIN,
                         ReadItemTimestamp + TimeZone(0));
    );

    Return(returnVal);
  ]
]
>

<
OPCGetItemAttributes
(
  itemID;
)
[
  retval;

  Constant OPCACCESS_READABLE      = 1;
  Constant OPCACCESS_WRITEABLE     = 2;
  Constant OPCACCESS_READWRITEABLE = 3;

  ItemData Struct [
    AccessRights      { Property ID number            };
    Type              { Data type of the property     };
    HasChildren;
  ];
]

Main [
  If watch(1);
  [
    IfThen(itemID == "tags\ai1",
      retval = ItemData(OPCACCESS_READABLE, VT_I4, 0);
   );
    IfThen(itemID == "tags\ai2",
      retval = ItemData(OPCACCESS_READABLE, VT_R8, 0);
    );
    IfThen(itemID == "tags\TextTag",
      retval = ItemData(OPCACCESS_READWRITEABLE, VT_BSTR, 0);
    );
    IfThen(itemID == "tags\ao1",
      retval = ItemData(OPCACCESS_READWRITEABLE, VT_R8, 0);
    );
    IfThen(itemID == "tags\ao2",
```

```
      retval = ItemData(OPCACCESS_READWRITEABLE, VT_I4, 0);
    );
    IfThen(itemID == "tags\AnalogInputs\ai1",
      retval = ItemData(OPCACCESS_READABLE, VT_I4, 0);
    );
    IfThen(itemID == "tags\NeverSet",
      retval = ItemData(OPCACCESS_READABLE, VT_BSTR, 0);
    );

{ The following elements are just structural - they have no data }
    IfThen(itemID == "", { A top-level name }
      retval = 0;
    );
    IfThen(itemID == "tags",
      retval = 0;
    );
    IfThen(itemID == "tags\AnalogInputs",
      retval = 0;
    );

    Return(retval);
  ]
]
>

{ End of SimulateOPCServer\CallbacksParent }
>

{ End of SimulateOPCServer }
>
```

Or

Description: Performs a bit-wise OR operation and returns the result.

Returns: 32 bit unsigned integer

Usage: Script or steady state.

Function Groups: Bitwise Operation

Related to:

Format: Or(Parm1, Parm2)

Parameters:

Parm1, Parm2

Required. Any numeric expressions. The expressions will be truncated to 32 bit unsigned integers. A and B must be in the range 0 to 4 294 967 294.

Comments: If either parameter is invalid, the return value is invalid.

Example:

```
newVal = Or(0b1010, 0b1100);
```

The value of newVal will be 0b1110.

Out

Description: Writes an 8 bit byte to an I/O port.

Returns: Nothing

Usage: Script or steady state.

Function Groups: Memory I/O

Related to: In | InWord | OutWord

Format: Out(, Value)

Parameters:

Port

Required. Any numeric expression giving the I/O address of the port to write. Port must be in the range 0 to 65535.

Value

Required. Any numeric expression giving the byte to write. Value must be in the range 0 to 255.

Comments: This function requires that the VTSIO driver be installed. Please refer to the topic, Communicating Directly With Hardware for more details.
If Port or Value is out of range or invalid, nothing is written. If this statement appears in a state, Value is written when its value changes. If it is in a script, Value is written when the script is executed.

Example:

```
Out(0x300, reg);
```

This writes reg to CPU output port 300 (hex).

Output

Note: Deprecated. Do not use in new code.

Description: Places formatted numbers or text on the screen.

Returns: Nothing

Usage: Steady State only. See: Rules for Usage.

Function Groups: Graphics

Related to: Format | GUIText | TextAttribs | ZText

Format: Output(X, Y, Type, Width, Precision, Value, Foreground, Fill, Background, Size, Obsolete)

Parameters:

X

Required. Any numeric expression giving the X screen coordinate of the lower left corner of the number or text on the screen.

Y

Required. Any numeric expression giving the Y screen coordinate of the lower left corner of the number or text on the screen.

Type

Required. Any numeric expression giving the type of the data to display. The valid values for this parameter are:

Type	Data Type
1	Short
2	Long
3	Float
4	Text
5	Binary
6	Octal
7	Hexadecimal

Status variables can be output using a Type value of 1 that will produce either a 0 or 1 on the screen. The type of the Value parameter does not have to match the Type parameter, except for text.

Width

Required. Any numeric expression giving the minimum number of characters to display. If fewer characters are required to produce the output, the area is filled with blank spaces on the left to make up the required number of characters. This is useful for aligning numbers up on the right. If more characters are required than the Width parameter specifies, the extra characters are extended to the right.

By making Width zero, the output will be aligned on the left. If the Width parameter is greater than or equal to 100 and the Type parameter is 3, the format of the floating point number displayed is in the most compact form which may be in exponential form if the exponent is less than -4 or is greater than the specified

Precision parameter.

The actual width used in this mode is 100 less than the specified width. Trailing zeroes are not displayed in this mode. Values of Width outside the range of 0 to 255 inclusive are invalid.

Precision

Required. Any numeric expression giving the precision of the output. This has different meanings for the different output types.

For types 1 and 2 (short and long), it gives the minimum number of digits to appear. If fewer digits are required to display the number, leading zeroes are added to the number.

For type 3 (float), it gives the number of digits to appear after the decimal point if Width is less than 100. If Width is greater than or equal to 100, it specifies the maximum number of significant digits to appear.

For type 4 (text), it gives the maximum number of characters to display. If the string is longer than this parameter, only the number of characters given by the Precision parameters are displayed.

Note: Values of Precision outside the range of 0 to 255 are invalid.

Value

Required. Any numeric or text expression giving the value to be displayed. If the Type is 4, this value may only be a text expression. Otherwise, this value is not required to have the same type as specified by the Type parameter – VTScada does the conversion.

Foreground

Required. Any numeric expression giving the color of the characters to be displayed.

Fill

Obsolete - set to zero.

Background

Required. Any numeric expression giving the color of the background area for the output characters.

Size

Required. Any numeric expression giving the height of the characters in units of Y screen coordinates. If this value results in a specification of less than 12 screen pixels high, the text will be the small text (8 pixels high). Otherwise, the text will be the large text. If Size is negative, it will be interpreted as a dot text output of size equivalent to the absolute value of the size. The number will be displayed to the nearest multiple of the base 8 pixel by 8 pixel text. This produces faster, non-destructive large characters than the normal large text characters.

Obsolete n/a

No longer used, but is maintained for backward compatibility with previous versions of VTScada. Set to 0.

Comments: Should not be used for new code. Maintained for backward-compatibility only. This statement is the general statement for displaying numbers and text on the screen. For small characters, or large characters with a negative Size, the statement is non-destructive (can change without destroying the underlying image). Large character output is destructive. For non-destructive out put, care must be taken when choosing the colors on a colored (non-zero) background since the characters are exclusive ORed with the background. This gives a more general method of displaying text than the Text statement but requires more parameters.

OutWord

Description: Writes a 16 bit word to an I/O port.

Returns: Nothing

Usage: Script or steady state.

Function Groups: Memory I/O

Related to: In | InWord | Out

Format: OutWord(Port, Value)

Parameters:

Port

Required. Any numeric expression giving the I/O address of the port to write. Port must be in the range 0 to 65535.

Value

Required. Any numeric expression giving the word to write. Value must be in the range 0 to 65535.

Comments: This function requires that the VTSIO driver be installed. Please refer to the topic, Communicating Directly With Hardware for more details.
This function writes a 16 bit unsigned value to a CPU I/O port. If this statement appears in a state, Value is written when its value changes. If it is in a script, Value is written when the script is executed.

Example:

```
OutWord(0x300, reg);
```

This writes reg to I/O port 300 (hex).

OwningModule

Description Returns the module which contains a certain variable.

Warning This function should be used by advanced users only.

Returns Module

Usage Script or steady state.

Function Groups Compilation and On-Line Modifications, Advanced Module

Related to: FindVariable

Format: OwningModule(Variable)

Parameters

Variable

Required. Any expression for the variable value.

Comments None

P Functions

The sections that follow identify all VTScada functions beginning with "P".

Pack

Description: Packs a set of module parameters or an array of values into a stream, and returns the number of items that were not packed.

Returns: Numeric

Usage: Script Only.

Function Groups: Advanced Module, Stream and Socket

Related to: Unpack | BuffStream | TempFileStream

Format: Pack(Data, Start, End, StreamVarPtr)

Parameters:

Data

Required. An object value, an array, or a structure.

Contains the data or the object value of the module whose parameters are to be packed.
For example, if you have 5 numeric values to pack, you would allocate a 1-dimensional array, 5 elements in length. You would then assign the 5 numerics to this array's elements, and pass the array to this parameter. You would specify that you wish to pack from subscript 1 to 5. Refer to the example section for more information.

Start

Required. The starting array index (zero-based), or parameter number (one-based) of the data to pack.

End

Required. For arrays this is the last array index. For parameters, it is the parameter number of the data to pack. If packing a structure rather than an array, this must be the number of elements rather than the final index.

StreamVarPtr

Required. A pointer to a variable holding the stream into which the data will be packed. The variable can also hold Invalid - see Comments section for more details.

Key

Optional. A short name or number. May be used to pack the data into a form that is significantly smaller than would otherwise be obtained. Note that integers will consume one quarter the space of text for this part of the packed data. There is no requirement for the Pack Key parameter and the Unpack MirrorKey parameter to have the same number of elements. All that is required is that the Unpack MirrorKey dictionary

has all of the structures that are in that particular packed stream. If the Key doesn't have the structure that was packed, the returned data is a simple array rather than a structure.

Comments:

Pack() is recursive. If the values contain arrays, then those are packed and so on.

StreamVar must be a pointer to a variable, not simply the name of a variable that holds a stream. This is because the Pack function will create a new stream under certain circumstances and store the packed data in it.

If the variable that StreamVarPtr addresses contains a stream, then the data will be packed into that stream.
If the variable that StreamVarPtr addresses contains Invalid, Pack will create a new buffer stream, pack the data into it and store the new stream in the variable pointed at by StreamVarPtr.
If a buffer stream is supplied, or Pack creates a buffer stream, it will be replaced by a temporary file stream if the output stream exceeds 2Mb. This provides a balance between performance and memory use.
If a stream is provided, packing starts at the current stream position. After packing, the stream current position is left at the end of the packed data.
Only variables of numeric, text, or stream types will be packed. Arrays of those types, up to and including 3 dimensions, will also be packed.

Any illegal values are packed as Invalid. Further, the Pack function has been designed in such a way that multidimensional arrays that have not had a value assigned can be handled. The correct number of Invalids for the unallocated array dimensions are written to the packed stream in place of the data values that would be there if the

elements were allocated. Once the stream is unpacked, the array elements are allocated, but are assigned Invalid. Generally called in the form:

```
Pack(Self(), 1, NParm(Self()), &Stream);
```

This works because NParm(Self()) equals the highest index.

Examples:

Parameters:

```
Pack(Self, 1, NParm(Self), &Stream); { Pack all parameters }
```

```
Pack(Self, 2, 2, &Stream); { Pack only the second parameter }
```

Arrays:

```
Pack(Array, 0, ArraySize(Array, 0) - 1, &Stream); { pack whole array
(0 based) }
```

```
Pack(Array, 1, 1, &Stream); { pack only the second array item }
```

Multi-dimensional array:

If you wish to pack 2 3-dimensional arrays, you would allocate a 1-dimensional array, 2 elements in length. You would then assign the 3-dimensional arrays to the 1-dimensional array elements, and then Pack the 1-dimensional array.

```
Array1D = New(2);
Array1D[0] = Array3D_1;
Array1D[1] = Array3D_2;
Pack(Array1D, 0, 1, &Stream);
```

Unpacking this is almost an identical operation:

```
Array1D = New(2);
UnPack(Array1D, 0, 1, Stream);
Array3D_1 = Array1D[0];
Array3D_2 = Array1D[1];
```

All you need to know is how many things you wish to pack and unpack, not the sizes or types of the things (a ValueType(Array1D[0]) will tell you the type). As the stream position is moved after each pack or unpack, you can simply unpack one item at a time and check the return value of UnPack.

Example 2:

In this example, the presence of a Key parameter allows the example to be packed into a stream that is less than half the size that would be obtained without the Key parameter. (Exact number depending on the names used in the structure definitions.)

"Record", "Values", and "AA_Info" are all STRUCTs.

```
Rec = Record( GetGUID(1)              { GUID       },
              1                       { Priority   },
              Values(87, 54, "feet") { Values     },
              System\MakeDictionary("AA",
                            AA_Info("AAData1", 2)) { Extensions });

{ Omitted: Place record in ArrayToPack }

Key = \System\MakeDictionary("Record", 0,
                             "Values", 1,
                             "AA_Info", "AA")

Pack(ArrayToPack, 0, 0, &PackStream, Key);

{ Omitted: Rewind stream }

MirrorKey = \System\MakeDictionary("0", Record,
                                   "1", Values,
                                   "AA", AA_Info));

UnPack(UnpackedArray, 0, 0, PackStream, Invalid, Invalid, MirrorKey);
```

PackParms

(RPC Manager Library)

Description: This method packs supplied parameters into a stream. Subroutine call only.

Returns: Nothing

Usage: Script Only.

Function Groups: Network, Stream and Socket

Related to: UnpackParms

Format: \RPCManager\PackParms(Stream [, P1, P2, ...]);

Parameters:

Stream

Required. The initial packed RPC stream. The parameters provided in parameters P1, P2, etc. will be packed and appended onto this stream. If Invalid, a new BuffStream will be created to hold the packed parameters.

P1, P2, ...

Optional parameters that will be packed into the stream. Up to 100 parameters are allowed.

Comments: This subroutine is a member of the RPC Manager's Library, and must therefore be prefaced by \RPCManager\, as shown in the "Format" section. If the application you are developing is a script application, the subroutine call must be prefaced by System\RPCManager\, and the System variable must be declared in AppRoot.src.
The stream contents can be unpacked into variables using the UnpackParms method. The parameters can be any of the permitted RPC data types. If a parameter is provided which is other than one of those data types, an Invalid is packed in its place.

PackRPC

(RPC Manager Library)

Description: Packs an RPC call and a set of parameters into a stream. Subroutine call only.

Returns: BuffStream

Usage: Script Only.

Function Groups: Network, Stream and Socket

Related to: RunPack

Format: \RPCManager\PackRPC(Stream, ModuleName, Scope [, Parameters]);

Parameters:

Stream

Required. The initial packed RPC stream. The RPC specified in this call will be appended onto the end of this stream. If Invalid, a new BuffStream will be created to hold the RPC.

ModuleName

Required. The textual name of the RPC subroutine to be executed. Must be valid.

ModuleContext

Required. The context in which the "ModuleName" will be executed. The "base" context for a VTScada layer-based application is "\Code". For a non-VTScada (pure script) application, the base context is "\System". Must be valid.

Parameters

Required. A set of up to 32 parameters to the RPC subroutine. Can be any mixture of the legal types. Supplying a parameter of an illegal type will cause it to be replaced with Invalid when the RPC subroutine is invoked.

Comments:

This subroutine is a member of the RPC Manager's Library, and must therefore be prefaced by \RPCManager\, as shown in the "Format" section. If the application you are developing is a script application, the subroutine call must be prefaced by System\RPCManager\, and the System variable must be declared in AppRoot.src.
The return value from this method is a BuffStream containing the original contents as specified in the Stream parameter with the new RPC appended. This can be used to build a stream of RPCs for transmission as one atomic unit.

PAddressEntry

(Dialog Library)

Description: To be used for Tag I/O address entry. Checks whether the attached driver has an AddressAssist module and uses that if available. Otherwise, presents a standard edit field into which the I/O address may be entered.

Returns: Nothing

Usage: Steady State only.

Function Groups: Graphics

Related to: GUITransform | PAreaSelect | PCheckBox | PContributor | PDroplist | PEditfield | PPageSelect | PRadioButtons | PSecBit | PSelectObject | PSpinbox | PTypeToggle

Format: \DialogLibrary\PAddressEntry(ParmNum, IODevice, SupportedData, FunctionType, Title, FocusID[, Trigger, DrawBevel, MinVal])

Parameters:

ParmNum

Any numeric expression giving the parameter number (from 0) in the caller to alter.

IODevice

Any expression for the object value of the I/O device driver being used.

SupportedData

A bitwise expression, indicating the data type.

Bit	Meaning when set
0	Digital
1	Analog
2	Text

FunctionType

Any Boolean expression, indicating whether the function should be read (0) or write (1).

Title

Any text expression to use as the title for the field.

FocusID

Boolean. If this value is FALSE (0), the field will display its current setting, but cannot be opened (i.e. its value cannot be changed), and will appear disabled (grayed-out).

Trigger

A parameter whose value is derived from ZEditField and can therefore be set to "0" (internal buffer changed), "1" ("Enter" key pressed), or "2" (focus lost). If this information is not required and the next parameter is used, a value of invalid or a constant may be substituted.

DrawBevel

Any logical expression. If TRUE, a bevel is drawn around the graphic.

MinVal

Optional. Any expression giving the minimum value or minimum number of characters to be accepted by the edit field (depending on the data type).
This value may be a decimal, octal or hexadecimal value. If this parameter is valid and a value less than LowLimit is entered in the field (or there are too few characters, in the case of text value), the variable set by the field will revert to the previous value. No default:

Comments: This module is a member of the VTScada Dialog Library and must therefore be called from within a GUITransform and prefaced by \DialogLibrary\.

The "P" tools (PCheckBox, PContributor, PColorSelect, PDroplist, and PEditfield) were intended only for use in configuration folders and drawing panel modules, and therefore are subject to the system security restraints.

PAddressEntry should not be used with the AnalogStatus tag as that tag stores Address, History Address and History Scan Rate together in one parameter. Use AddressEntry instead.
Usual height of the GUITransform: 45 - 55 pixels.

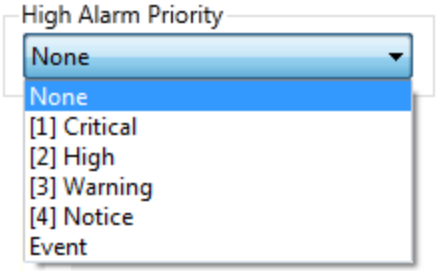
Example:

```
{***** Address *****}
GUITransform(30, 160, 230, 115,
  1, 1, 1, 1, 1 { No scaling },
  0, 0, 1, 0 { No movement; visible; reserved },
  0, 0, 0 { Not selectable },
  \DialogLibrary\PAddressEntry(\#Address { Parm number },
          Scope(\Root, Parms[\#SitePoint]) { IO Device },
          0b010 { SupportedData: Analog },
          1 { FunctionType: Write },
          \AddressLabel { Title },
          3 { Focus ID },
          Trigger { trigger }));
```

PAlmPriority

(Dialog Library)

Description: Draws a droplist of the currently available alarm priorities with an optional title and bevel.



Returns: Image handle

Usage: Steady State only.

Function Groups: Alarm

Related to: GUITransform | PAreaSelect | PCheckBox | PColorSelect | PContributor | PDroplist | PEditfield | PPageSelect | PRadioButtons | PSecBit | PSelectObject | PSpinbox | PTypeToggle

Format: \DialogLibrary\PAlmPriority(ParmNum, Title[, AlignTitle, FocusID, Trigger, Init, DrawBevel, VertAlign])

Parameters:

ParmNum

Required. Any numeric expression giving the parameter number (from 0) in the caller to alter.

Title

Required. Any text expression to be used as a title for the droplist.

AlignTitle

Optional. Any logical expression. Indicates the title alignment such that if it is true (non-zero), the title is included in the calculation for vertical alignment. If false (0), it is added to the droplist after it, and its bevel (if one exists) has been vertically aligned.
The default is true.

FocusID

Boolean. If this value is FALSE (0), the field will display its current setting, but cannot be opened (i.e. its value cannot be changed), and will appear disabled (grayed-out).

Trigger

Optional. Any logical expression. A variable whose value is set to true (1) when the ParmNum parameter's

value has been set by the droplist. The setting of the parameter will not cause Trigger to be set.
If this information is not required, a constant may be used.

Init

Optional. Any expression for the initial value displayed in the field. The default value is Data[Index]. If Index is invalid, then the droplist will initially appear blank.

DrawBevel

Optional. Any logical expression. If true (non-0), a bevel is drawn around the droplist. If false (0), no bevel is drawn. The default value is true.

VertAlign

Optional. Any numeric expression that sets the vertical alignment of the unopened droplist according to one of the following:

Value	Vertical Alignment
0	Top
1	Center
2	Bottom

Whether or not the title is included when the vertical alignment is calculated is determined by the value of AlignTitle.
The default value is 0.

Comments: This module is a member of the VTScada Dialog Library, and must therefore be called from within a GUITransform and prefaced by \DialogLibrary\.
The "P" tools are intended only for use in configuration folders and drawing panel modules, and therefore are subject to the system security restraints.

This parameter tool expects the first parameter of its calling module to contain an array of tag parameters. It will then set the element indicated by ParmNum to the text string displayed in the selected line of the droplist.
The height of the unopened droplist is constant, with the horizontal boundaries of its calling transform defining its width, and the vertical boundaries of its calling transform defining its opened height, which will include the added height of the bevel above the field, but may or may not include the title, depending on the alignment used. Note that if the entire list can be displayed in a smaller area than indicated by the vertical boundaries of the calling transform, the dropped list height will be decreased. The dropped height of the list will always have a minimum height of 1 line (below the field).
Usual height: 45 pixels.

Example:

```
GUITransform(30, 200, Width/2 - 5, 155
  1, 1, 1, 1, 1 { No scaling       },
  0, 0, 1, 0  { No movement; visible; reserved  },
  0, 0, 0  { Not selectable    },
  \DialogLibrary\PAlmPriority(\#Priority {Parm to edit  },
       \PriorityLabel { Label },
       0    { Align Title },
       5   { ID  },
       1   { Trigger },
       0   { Init  },
       1   { Draw bevel },
       1   { Vert align }));
```

PalStatus

Description: Returns the color intensities of the current palette. Maintained for backward compatibility only.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Color, Graphics

Related to:

Format: PalStatus(Color, RGBIndex)

Parameters:

Color

Required. Any expression giving the color to examine. May be a number from 0 to 255 (from the VTScada palette) or an RGB value with alpha channel, expressed as "<AARRGGBB>".

RGBIndex

Required. Any numeric expression giving the color component to return. Red, green, blue, or alpha.

RGBIndex	Color Content
0	Red
1	Green
2	Blue
3	Alpha channel

Comments: The returned number ranges from 0 to 1, and indicates the red, green, or blue color content of a color index, or the alpha channel (opacity) in the case of index 3.

Example:

```
ZText(10, 10, Concat("Red = " , PalStatus(14, 0)), 14, 0);
ZText(10, 20, Concat("Green = ", PalStatus(14, 1)), 14, 0);
ZText(10, 30, Concat("Blue = " , PalStatus(14, 2)), 14, 0);
```

This displays the color mix for color index 14 (yellow – FFFFFF00) in that color. The values displayed in this case will be 1, 1 and 0 respectively.

Parameter

Description: Returns the value of (or may assign a value to) a parameter of a module, specified by the index.

Returns: varies

Usage: Script Only.

Function Groups: Basic Module, Variable

Related to: BuffToParm | NParm | ParmToBuff | PType | ResetParm

Format: Parameter(Object, Index)

Parameters:

Object

Required. Any object (the object value of any module instance).

Index

Required. Any numeric expression giving the number of the parameter of interest, starting from 1.

Comments: This function returns the value passed in the module call in the position Index. It may also appear on the left side of an assignment operator (=), in which case it will try to assign the value to the parameter in Index. The parameter must be a reference type parameter to receive an assignment (all undeclared parameters default to reference type). Assignment to a non-reference parameter or function has no effect. The return value of Parameter is invalid if there is no parameter at Index.
For launched modules, this function will not look at parameters beyond the number of formal parameters.

Example:

```
If 1 Main;
[
  Parameter(Self(), 5) = 4;
]
```

This attempts to set parameter #5 to 4. Either the module must have parameter 5 declared as a reference parameter, or have no declaration for parameter 5 at all. The module call must have at least 5 parameters listed, and the fifth parameter must be a variable or array element. If any of these conditions are not met, nothing happens. If the module call is:

```
Show Normalize(1, 2, 3, 4, X);
```

Then X is set to 4.

ParameterEdit

Description: Draws an interface to allow the user to choose how to edit a parameter. Used in tag widgets.

Returns: Self

Usage: Steady State only.

Function Groups: Graphics, Variable

Related to: NumericParameterEdit | ParameterSet

Format: \ParameterEdit(ParmVal, ParmPtr, Enable, Title, Modules, Contexts, Parameters, TitleWidth, StartIndex, PtrWaitClose, DialogRoot[, FocusID, Description])

Parameters:

ParmVal

The variable to be changed. Metadata indicating the revision number (always 0) must be attached.

ParmPtr

Code Pointer to the parameter value so the Para-meterEdit modules can dissect and categorize it

Enable

Flag - TRUE to enable the drawing of this parameter edit module.

Title

Title for this parameter

Modules

Array of module names (snap-ins such as ParmEditColor, etc.) for parameter editing

Contexts

Contexts of where to find the Parameter edit modules (usually, \Code)

Parameters

Parameters for the parameter edit modules. A multidimensional array, where each sub-array is a parameter list for each of the entries in the Modules array.

TitleWidth

Width allotted for the title. If Invalid, uses a standard size (160px).

StartIndex

Starting Index for Parm Edit Modules

PtrWaitClose

Set to true to tell caller to wait to close

DialogRoot

Calling dialog window

FocusID

Optional focus ID value. Defaults to 1 if the control is enabled, otherwise 0. The parameter edit control will have two parts and thus use two ID values: the one specified (or the default) and one greater than that.

Description

Optional text, describing this parameter

Comments:

Metadata indicating the version number must be added to the ParmVal parameter. For example,

```
{ Set up variables for ParameterEdit }
    MetaData(Value, "Revision") = 0;
```

Wrappers such as NumericParameterEdit exist to

make this function easier to use. See: ParameterEdit Snap-ins

Example:

```
GUITransform(0, 1, 1, 0                          { Unit Outline
},
             1 - (Left)                          { Left Scaling
},
             Top + PickValid(Height, 0)          { Bottom Scaling
},
             Right                               { Right Scaling
},
             1 - (Top + TitleBarHeight)          { Top Scaling
},
             1                                   { Overall Scaling
},
             0, 0                                { Trajectory, Rotation
},
             PickValid(DrawOptions, 1), 0        { Visibility, Reserved
},
             0, 0, 0                             { Selectability
},
             ParmEditObj = \ParameterEdit(
                             TextHAlign          { Parameter Value
},
                             CodePtr             { Pointer to Parm Code
},
                             PickValid(DrawOptions, 1) { Enable Flag
},
                             PickValid(ParmLabel, \Hor-
izAlignmentLabel)
                                                                 {
Title },
                             Modules             { Array of Parm Edit Mod-
ules },
                             Contexts            { Contexts for Edit
Modules  },
                             Parameters          { Parameters for Edit
Modules},
                             Invalid             { Title Width
},
                             Invalid             { Start Index
},
                             PtrWaitClose        { Wait to close
},
                             DialogRoot          { Calling dialog window
}
                           ));
```

Related Information:

ParameterEdit Snap ins

ParameterSet

Description: When passed a set of up to 256 parameters, will return them as an array.

Returns: Array

Usage: Steady State only.

Function Groups: Array, Variable

Related to: ParameterEdit

Format: \ParameterSet(Parm0[Parm1, ... Parm256])

Parameters:

Parm0

The first parameter to include.

Parm1 to Parm256

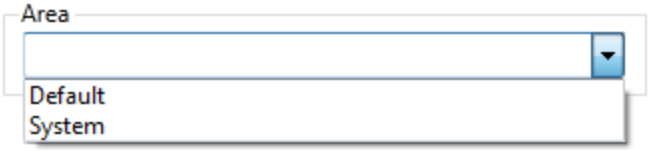
More parameters.

Comments: The returned array of parameters is suitable for use with ParameterEdit.

PAreaSelect

(Dialog Library)

Description: Draws a droplist of area options with optional title and bevel.



Returns: Nothing

Usage: Steady State only.

Function Groups: Graphics

Related to: GUITransform | PAddressEntry | PCheckBox | PColorSelect | PContributor | PDroplist | PEditfield |

PPageSelect | PRadioButtons | PSecBit | PSelectObject | PSpinbox | PTypeToggle

Format: \DialogLibrary\PAreaSelect(CanEdit, FocusID, Init, DrawBevel, VertAlign, AlignTitle, ParmNum, ExtTrigger, OKPressed)

Parameters:

CanEdit

Any logical expression. If true (non-0) the text displayed in the area droplist can be edited in the same manner as an editfield, if false (0) it cannot be edited directly. Defaults to true (1) if invalid.

FocusID

Boolean. If this value is FALSE (0), the field will display its current setting, but cannot be opened (i.e. its value cannot be changed), and will appear disabled (grayed-out).

Init

Any numeric expression indicating the initial value. No default is provided.

DrawBevel

Any logical expression. If true (non-0) a bevel is drawn around the area droplist, if false (0) no bevel is drawn. Defaults to true (1) if invalid.

VertAlign

Any numeric expression that sets the vertical alignment of the unopened area droplist according to one of the following:

VertAlign	Vertical Alignment
0	Top
1	Center
2	Bottom

Whether or not the title is included when the vertical alignment is calculated is determined by the value of AlignTitle. Defaults to 0 if invalid.

AlignTitle

Any logical expression. If true (non-0) the title is included in the calculation for vertical alignment. If false(0) it is added to the area droplist after it (and its bevel if one exists) has been vertically aligned. Defaults to true (1) if invalid.

ParmNum

A parameter number (starting from 0), that will be edited instead of the default Area parameter. Specify the parameter to which you wish to add an area.

ExtTrigger

An external trigger, to be set when the area changes. Output value only.

OKPressed

Required. Linked to the OK button on the Config folder.

Comments

This module is a member of the VTScada Dialog Library and must therefore be called from within a GUITransform and prefaced by \DialogLibrary\.
This parameter tool expects the first parameter of its calling module to contain an array of tag parameters. It will then set the second element (element 1) to the text string that designates the selected area.
The height of the (unopened) area droplist is constant, with the horizontal boundaries of its calling transform defining its width, and the vertical boundaries of its calling transform defining its opened height, which will include the added height of a

bevel, but may or may not include the title, depending on the alignment used. Note that if the entire list can be displayed in a smaller area than indicated by the vertical boundaries of the calling transform, the dropped list height will be decreased. The dropped height of the list will always have a minimum height of 1 line (below the field).
Usual height: 100 pixels.

Examples:

```
GUITransform(30, 180, 210, 80,
             1, 1, 1, 1, 1,
             0, 0, 1, 0,
             0, 0, 0,
             \DialogLibrary\PAreaSelect(0 { Not editable },
             1 { Focus ID },
             0 { No bevel },
             Invalid
             { Use default },
             0 { Align bevel }));
GUITransform(10, 120, 110, 20,
             1, 1, 1, 1, 1,
             0, 0, 1, 0,
             0, 0, 0,
             \DialogLibrary\PAreaSelect());
GUITransform(10, 120, 110, 20,
             1, 1, 1, 1, 1,
             0, 0, 1, 0,
             0, 0, 0,
             \DialogLibrary\PAreaSelect(1 { Not editable },
             2 { Focus ID },
             1 { Draw bevel },
             0 { Top align },
             1 { Align title }));
```

Notice that the last two examples produce the exact same results, because the last duplicates the default settings.

ParentModule

Description: Returns the parent module of a module.

Returns: Module

Usage: Script or steady state.

Function Groups: Advanced Module

Related to: Caller | ParentObject

Format: ParentModule(Module)

Parameters:

Module

Required. Any expression for the module.

Comments: The parent module is the module in which Module is defined.

Example:

```
ZText(10, 20, ParentModule(Self()), 15, 0);
```

This will display the parent module of the current module on the screen.

ParentObject

Description: Returns the parent of an object.

Returns: Module

Usage: Script or steady state.

Function Groups: Advanced Module

Related to: Caller | ParentModule

Format: ParentObject(Object)

Parameters:

Object

Required. The object you are inquiring about.

Comments: Returns the parent object of the given object instance.

Example:

```
parent = ParentObject(Self());
If ! Valid(Parent);
[
```

```
  slay(self(), 0);
]
```

This example illustrates how the function may be used to monitor a module's parent to prevent orphaning of the object.

ParentWindow

Description: Returns the object value of the nearest non-child window.

Returns: Window Object

Usage: Script or steady state.

Function Groups: Graphics, Window

Related to: ActiveWindow | RootWindow

Format: ParentWindow(Object)

Parameters:

Object

Required. Any object (the object value of any running module instance).

Comments: This function will start at the object value given and find the first window which is not a child window. If there is no window found the return value will be invalid.

For modules in non-child windows (i.e. one without bit 9 set), RootWindow and ParentWindow will return the same value.
For child windows, RootWindow will return the root module in the child window, while ParentWindow will return the root module in the child window's closest non-child calling window.
To test whether a custom widget is being displayed while in "editing mode" (that is, in the Idea Studio) you can test whether ParentWindow()\Editing is

TRUE or FALSE.

ParmToBuff

Description: Returns a buffer of formatted numeric parameter values.

Returns: Buffer

Usage: Script Only.

Function Groups: Advanced Module, String and Buffer

Related to: ArrayToBuff | BuffToArray | BuffToParm

Format: ParmToBuff(Object, Index, N, Option, Size, Skip [, BadData])

Parameters:

Object

Required. Any object (the object value of any running module instance).

Index

Required. Any numeric expression giving the first parameter to format, starting from 1.

N

Required. Any numeric expression giving the number of parameters to format. If there are fewer actual parameters than N + Index, this statement stops at the last parameter.

Option

Required. Any numeric expression which specifies the format of the buffer read.

Option	Format
0	Unsigned binary (low byte first)
1	Signed binary (low byte first)
2	BCD (binary coded decimal) (low byte first)
3	ASCII octal (high byte first)
4	ASCII decimal (high byte first)
5	ASCII hex (high byte first)
6	ASCII floating point (high byte first)
7	IEEE float/double (low byte first)
8	<obsolete>
9	Allen-Bradley® PLC/3 floating point
10	VAX single precision floating point

Size

Required. Any numeric expression giving the number of digits in each datum. It has a different meaning for each option as indicated:

Option	Size Meaning	Size Range
Binary types	Number of bits	1 - 32 bits
BCD	Number of 4-bit digits	1 - 8 digits
ASCII types	Number of bytes	1 - 32 bytes
Float types	Precision	1 for single precision, 2 for double precision

For Options 7 and 9 the data is written as appropriate binary format.

Skip

Required. Any numeric expression giving the number of buffer bits/digits/bytes to skip after writing each non-floating point element. For floating point types, this parameter must be set to 0.

BadData

An optional parameter that designates how invalid data is to be handled, according to the following table: Defaults to 0 if missing or invalid.

BadData	Handled
0	Output to buffer as invalid values
1	Causes buffer to be invalid
2	Output to buffer as valid 0s

Comments: This function may only be used with parameters containing numeric data. It is useful for encoding serial port data when writing I/O drivers.

Example:

If a module call looks like:

```
Write(1, 2, 3, x, y, z);
```

And there is a statement in module Write that looks like:

```
If ! Valid(writePacket);
[
  writePacket = ParmToBuff(Self(){ Current module },
                           4 { Skip first 3 parameters },
                            NParm(Self()) - 3
                            { Use rest of parameters },
                            0 { Unsigned binary format },
                            16 { Bits },
                            0 { No skip });
]
```

Since this statement encodes x, y, and z as 16 bit unsigned integers, the returned buffer will be 16 bytes long, byte-ordered as follows:

Byte	Description
0	Low byte of x
1	High byte of x
2	Low byte of y
3	High byte of y
4	Low byte of z
5	High byte of z

ParserSRO

Description: Adds a scope resolution reference to a variable on the top of the PARSER_STACK given the stack and the object variable.

Warning: This function should be used by advanced users only since irrevocable alteration of your application may occur.

Returns: Nothing

Usage: Script Only.

Function Groups: Compilation and On-Line Modifications

Related to: Compile

Format: ParserSRO(ParserStack, Variable)

Parameters:

ParserStack

Required. Any expression for the parser stack value.

Variable

Required. Any expression for the variable value that will be referenced by scope resolution.

Comments: The ParserStack is returned from the Compile function. This assumes that the text value for the variable after the \ is already added as a parameter. This is only used when

compiling text and is not required for normal operation.

PasteObjects

Description: Pastes the code for multiple GUITransforms into a page source file.

Returns: Nothing

Usage: Script Only.

Function Groups: Graphics

Related to: CopyObjects

Format: PasteObjects(ObjectText, SelDAGs)

Parameters:

ObjectText

Required. The text string containing objects.

SelDAGs

Required. An array of code pointers of the pasted objects.

Path

Description: Returns a graphics path value.

Returns: Path

Usage: Steady State only.

Function Groups: Graphics

Related to: GUIPolygon | Point | Trajectory | Vertex

Format: Path(Closed, V1, V2, ...)

Parameters:

Closed

Required. Any logical expression. If true, this is a closed path, and the last Vertex is considered to be

connected to the first point. If false, this is an open path ending at the last Vertex.

V1, V2, ...

Required. Any expressions that return Vertex values.

Comments: Paths are used to determine the shape of polygons, and as trajectories (animation motion paths).

Example:

```
truckPath = Path(0 { Open path, doesn't close on self },
                 Vertex(0 { Rectangular mode },
                 Point(20, 200, Invalid, Invalid),
                 Point(20, 200, Invalid, Invalid),
                 Point(20, 200, Invalid, Invalid)),
                 Vertex(0 { Rectangular mode },
                 Point(200, 200, Invalid, Invalid),
                 Point(200, 200, Invalid, Invalid),
                 Point(200, 200, Invalid, Invalid)),
                 endVertex { A variable containing a vertex });
```

This shows a 3 vertex path with one vertex found in a variable. This vertex might be used in other functions, such as another path.

PathDraw

Description: Used within a Idea Studio handler to inform the engine that a path is being drawn, and to set the appearance of that path. Used for both lines and pipes. The resulting object will be a GUIPolygon.

Returns: Numeric

Usage: Steady State only.

Function Groups: Graphics

Related to:

Format: PathDraw(Pen, Brush);

Parameters:

Pen

Required. Defines the color, style and width of

the path. May be any of the following:

Value	Notes
Pen object	If the pen color is transparent, invisible, or has a zero width, then a 1-pixel black line will be shown only while the path is being drawn to show the position.
Palette index	Defines only the color of the line. A 1-pixel, solid line will be drawn.
System color	Defines only the color of the line. A 1-pixel, solid line will be drawn.
-1 (transparent line)	The line will not be shown, except that a 1-pixel solid black line will be used while the path is being drawn.
aRGB string	Defines only the color of the line. A 1-pixel, solid line will be drawn.

Brush

Required. Defines the fill color of a closed figure or the color of a pipe. May be any of the following:

Value	Notes
Brush object	Defines a foreground, background and pattern for the fill.
Palette index	Defines only the color of the fill.
System color	Defines only the color of the fill.
-1 (transparent line)	The fill will not be shown.

aRGB string	Defines only the color of the fill.

Comments: Do not confuse PathDraw with the function DrawPath().
This function will work only inside the context of a window with the Drop Target bit (22) set.
The path may have multiple vertices. If the function stops during drawing, no portion will be drawn.
When drawing a pipe, set the style for the Pen() to 100.

Examples:

Render a path as a narrow black line:

```
PathDraw(Pen(("<FFFFFFFF>", 1, 1), Brush(("<FF000000>", 0, 1));
```

Render a path as a pipe:

```
PathDraw(Pen("<FF80FF80>", 100, 29), Brush(-1, 0, 1));
```

PatternMatch

Description: Compares a string against a reference pattern and returns true if the string matches the pattern. Along with literal characters, PatternMatch currently supports the * and ? wildcard characters within the reference pattern (see Pattern parameter).

Returns: Boolean

Usage: Script only.
May be used in optimized Tag Parameter Expressions.

Function Groups: String and Buffer

Related to:

Format: PatternMatch(String, Pattern [, CaseInsensitive])

Parameters:

String

Required. The string you wish to compare to Pattern.

Pattern

Required. The reference pattern to which you wish String to be compared. PatternMatch currently supports the * and ? wildcard characters in the reference pattern.

These wildcards may be escaped with a backslash (\), as can the backslash character itself. The backslash does not have to be escaped, so a lone backslash (that does not precede a * or ? character) is interpreted just like a double one: as a literal backslash.

CaseInsensitive

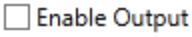
An optional Boolean parameter that indicates whether or not PatternMatch should be case insensitive when it performs the matching. The default for this parameter is FALSE, indicating that PatternMatch should be case sensitive.

Comments: If any of the arguments to PatternMatch are Invalid, PatternMatch returns Invalid.

PCheckBox

(Dialog Library)

Description: Parameter Setting check box. This module draws a check box with optional label.



Returns Nothing

Usage:: Steady State only.

Function Groups: Graphics

Related to: GUITransform | PAddressEntry | PAreaSelect | PColorSelect | PContributor | PDroplist | PEditfield |

PPageSelect | PRadioButtons | PSecBit | PSelectObject | PSpinbox | PTypeToggle

Format: \DialogLibrary\PCheckBox(ParmNum [, Label, BoxOnLeft, Alignment, FocusID, PrivNotReq])

Parameters:

ParmNum

Required. Any numeric expression giving the parameter number (from 0) in the caller to alter.

Label

An optional parameter that is any text expression to be used as a label with the check box. The default value is a blank label.

BoxOnLeft

An optional parameter that is any logical expression. If true (non-0) the check box will appear to the left of the label, if false (0) it will be to the right. The default value is true.

Alignment

An optional parameter that is any numeric expression that sets the alignment of the check box and its label according to one of the following options:

Alignment	Horizontal Alignment	Vertical Alignment
0	Left	Top
1	Right	Top
2	Full	Top
3	Left	Centered
4	Right	Centered
5	Full	Centered
6	Left	Bottom
7	Right	Bottom
8	Full	Bottom

FocusID

Boolean. If this value is FALSE (0), the field will display its current setting, but cannot be opened (i.e. its value cannot be changed), and will appear disabled (grayed-out).

PrivNotReq

An optional parameter that is any logical expression. If set to true (non-0), anyone can change the value in this editfield. If set to false (0), only those users whose user accounts have been granted the "TagModify" priv-

ilege may set it. The default value is false.

Comments: This module is a member of the VTScada Dialog Library and must therefore be called from within a GUITransform and prefaced by \DialogLibrary\.
The "P" tools (Pcheck box, PContributor, PColorSelect, PDroplist, and PEditfield) were intended only for use in configuration folders and drawing panel modules, and therefore are subject to the system security restraints.
This parameter tool expects the first parameter of its calling module to contain an array of tag parameters. It will then set the element indicated by ParmNum to the logical value set by the check box.
The size of the check box is constant, with the boundaries of its calling transform defining the position of the check box and its label.
Usual height: 12 pixels.

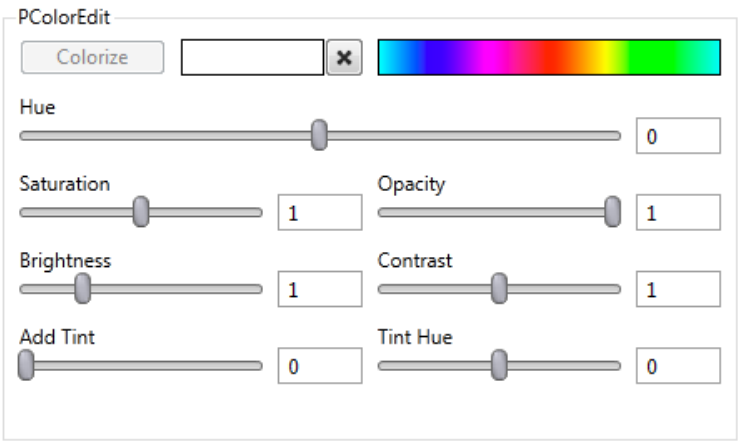
Examples:

```
GUITransform(70, 190, 210, 170,
             1, 1, 1, 1, 1,
             0, 0, 1, 0,
             0, 0, 0,
             \DialogLibrary\PCheckBox(3 { Parm num },
             "Invert Output" { Label },
             0 { Box on right },
             2 { Full, top alignment },
             4 { Focus ID }));
```

PColorEdit

(Dialog Library)

Description: Wrapper module for the standard Color editing Parameters including Hue, Saturation, Brightness, Transparency, Contrast and Colorizing Hue and Intensity.



Returns: Nothing

Usage: Steady State only.

Function Groups: Graphics

Related to: GUITransform | PAddressEntry | PAreaSelect | PColorSelect | PContributor | PDroplist | PEditfield | PHueSelect | PPageSelect | PRadioButtons | PSecBit | PSelectObject | PSpinbox | PTypeToggle

Format: \DialogLibrary\PColorEdit(Parms, Hue, Saturation, Brightness, Transparency, Contrast, ColorizeHue, ColorizeInt[, Title, FocusID, Trigger])

Parameters:

Parms

Required. A pointer to the array of parameters.

Hue

The parameter number for the Hue.

Saturation

The parameter number for the Saturation.

Brightness

The parameter number for the Brightness.

Transparency

The parameter number for the Transparency.

Contrast

The parameter number for the Contrast.

ColorizeHue

The parameter number for the ColorizeHue.

ColorizeInt

The parameter number for the Colorize Intensity.

Title

An optional parameter that is any expression for a title to be applied.

FocusID

Boolean. If this value is FALSE (0), the field will display its current setting, but cannot be opened (i.e. its value cannot be changed), and will appear disabled (grayed-out).

Trigger

Set when the variable is changed.

Comments:

This module is a member of the VTScada Dialog Library and must therefore be called from within a GUITransform and prefaced by \DialogLibrary\.
The "P" tools (PCheckBox, PContributor, PColorSelect, PDroplist, and PEditfield) were intended only for use in configuration folders and drawing panel modules, and therefore are subject to the system security restraints.
This parameter tool expects the first parameter of its calling module to contain a pointer to an array of tag parameters. It will then set the elements indicated by Hue, Saturation, Brightness, Transparency, Contrast, ColorizeHue and ColorizeInt to the logical value set by the matching input field.
This module provides an easier way to add color

editing to a Panel module. It will default invalid parameters.
Usual height: 260 pixels.

Note: This module uses 7 Focus ID's.

Examples:

```
  GUITransform(0, 1, 1, 0,
               1                                  { Scale Left    },
               PanelHt - TitleSpace               { Scale Bottom  },
               PanelWd                            { Scale Right   },
               1                                  { Scale Top     },
               1                                  { Scale Whole   },
               0, 0, 1, 0          { No movement; visible; reserved
},
               0, 0, 0             { Not selectable
},
               \DialogLibrary\PColorEdit(Parms,
                                   #Hue,
                                   #Saturation,
                                   #Lightness,
                                   #Transparency,
                                   #Contrast,
                                   #ColorizeHue,
                                   #ColorizeIntensity,
                                   \ColorOptionsLabel { Title
},
                                   9          { to 15 - ID
},
                                   Trigger    { Trigger
}));
```

PColorSelect

(Dialog Library)

Description: Draws a button that opens a color selection dialog and an area that displays the currently selected color.

Returns: Nothing

Usage: Steady State only.

Function Groups: Color, Graphics

Related to: GUITransform | PAddressEntry | PAreaSelect | PCheckBox | PContributor | PDroplist | PEditfield | PPageSelect | PRadioButtons | PSecBit | PSelectObject | PSpinbox | PTypeToggle

Format: \DialogLibrary\PColorSelect(ParmNum [, BtnLabel, BtnOnLeft, Standard, VertAlign, FocusID, Open])

Parameters:

ParmNum

Required. Any numeric expression giving the parameter number (from 0) in the caller to alter.

BtnLabel

An optional parameter that is any text expression to be used as a label on the color selection button.

BtnOnLeft

An optional parameter that is any logical expression. If true (non-0) the button will appear to the left of the color display area, if false (0) it will be to the right. The default value is true.

Standard

An optional parameter that is any logical expression. If true (non-0) the button and color display area will be standard system size, if false (0) they will be sized to fit their boundaries and VertAlign will be ignored. The default value is true.

VertAlign

An optional parameter that is any numeric expression that sets the vertical alignment of the button and dis-

VertAlign	Vertical Alignment
0	Top
1	Center
2	Bottom

Note: If Standard is true, this parameter is ignored. The default value is 0, top alignment.

FocusID

Boolean. If this value is FALSE (0), the field will display its current setting, but cannot be opened (i.e. its value cannot be changed), and will appear disabled (grayed-out).

Open

An optional parameter that is any logical expression. If true (non-0) the dialog will be open. If false (0), it will be closed. The default is true.

Comments:

This module is a member of the VTScada Dialog Library and must therefore be called from within a GUITransform and prefaced by \DialogLibrary\.
The "P" tools (PCheckBox, PContributor, PColorSelect, PDroplist, and PEditfield) were intended only for use in configuration folders and drawing panel modules, and therefore are subject to the system security restraints.
This parameter tool expects the first parameter of its calling module to contain an array of tag parameters. It will then set the element indicated by ParmNum to the chosen color index.
If the button and display area are set to be standard size, the button size will remain a constant size (101 x 31 pixels) regardless of the calling transform, but the display area will vary between a minimum (square) size to a maximum length equal to the length of the button, depending on the transform. Once the maximum display area size has been reached, the button and area will be fully justified to cover the entire width of the transform (the ver-

tical alignment will be set by the value of VertAlign). For any optional parameter that is to be set, all optional parameters preceding the desired one must be present, although they may be invalid. Usual height: 21 pixels.

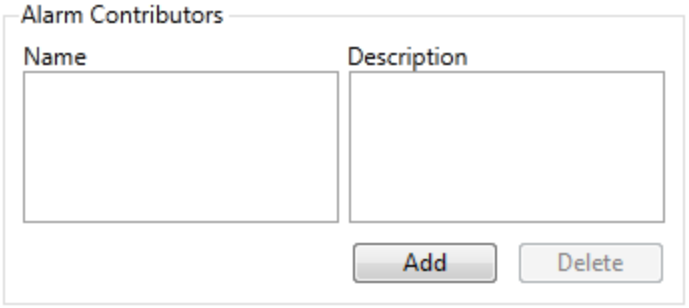
Example:

```
GUITransform(70, 200, 290, 170,
             1, 1, 1, 1, 1,
             0, 0, 1, 0,
             0, 0, 0,
             \DialogLibrary\PColorSelect(8 { Parm num },
             "Set Color" { Label },
             1 { Button on left },
             1 { Standard size },
             1 { Centered vertically },
             4 { Focus ID }));
```

PContributor

(Dialog Library)

Description: Draws a split list displaying all contributors to a specific tag.



Returns: Nothing

Usage: Steady State only.

Function Groups: Containers and Contributors, Graphics

Related to: AddContributor | GetContributors | GUITransform | PAddressEntry | PAreaSelect | PCheckBox | PColorSelect | PDroplist | PEditfield | PPageSelect | PRadioButtons |

Format: \DialogLibrary\PContributor(HolderName, ContainerObj, ContribType, TagType [, Title, FocusID, NoAdd])

Parameters:

HolderName

Required. Any text expression giving the variable name for container information.

ContainerObj

Required. The object value of the container.

ContribType

Required. Any text expression giving the contribution type.

TagType

Required. Any text expression for the type of Tag for which the contributors are being sought.

Title

An optional parameter that is any text expression to be used as a title for the splitlist.

FocusID

Required. A parameter that is any numeric expression for the focus number of the splitlist and its two buttons. The splitlist will have a focus ID equal to FocusID, the left button will have one of FocusID + 1, while the right button will have a focus ID of FocusID + 2.
If FocusID is 0, the splitlist will display its current setting, but its accompanying buttons will not be able to be selected.

NoAdd

An Boolean optional expression. Set TRUE when the Add button is disabled.

Comments:

This module is a member of the VTScada Dialog Library and must therefore be called from within a GUITransform and prefaced by \DialogLibrary\.
The "P" tools (PCheckBox, PContributor, PColorSelect, PDroplist, and PEditfield) were intended only for use in configuration folders and drawing panel modules, and therefore are subject to the system security restraints.
This parameter tool expects the first parameter of its calling module to contain an array of tag parameters. It will then set the element indicated by ParmNum to the chosen tag's name.
For any optional parameter that is to be set, all optional parameters preceding the desired one must be present, although they may be invalid.
This function provides a split list containing the names and descriptions of all contributors stored in HolderName. Double-click on an item in this list to obtain the Properties dialog for the tag. Click on the Add button to add a new contributor of the tag type "PointType", which is owned by "ContainerObj", and which has a contribution type "ContribType". Click on the Delete button to terminate the container/contributor relationship for the currently highlighted contributor. The former contributor is not deleted from the application, or from the database.
Usual height: 200-250 pixels.

Examples:

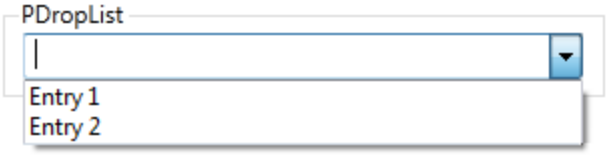
```
GUITransform(10, 410, 310, 10,
             1, 1, 1, 1, 1,
             0, 0, 1, 0,
             0, 0, 0,
             \DialogLibrary\PContributor(
```

```
            "" { Holder name },
            Var { Container object },
            "" { Contributor type },
            "AnalogInput" { Tag type },
            "Contributors" { Title },
            4 { Focus ID }));
```

PDroplist

(Dialog Library)

Description: Parameter Setting Droplist. This module draws a droplist with (optional) title and bevel.



Returns: Nothing

Usage: Steady State only.

Function Groups: Graphics

Related to: GUITransform | PAddressEntry | PAreaSelect | PCheckBox | PColorSelect | PContributor | PEditfield | PPageSelect | PRadioButtons | PSecBit | PSelectObject | PSpinbox | PTypeToggle

Format: \DialogLibrary\PDropList(ParmNum, Title, DataOrA-lignTitle [, CanEdit, Index, FocusID, Trigger, Init, DrawBevel, VertAlign, AlignTitle_L0, RetArray_L1, L2, ..., L16])

Parameters:

ParmNum

Required. Any numeric expression giving the para-meter number (from 0) in the caller to alter.

Title

Required. Any text expression to be used as a title for the droplist.

DataOrAlignTitle

Required. A flexible parameter that is either an array of data to be displayed in the droplist, or a logical expression indicating the alignment of the droplist's title.
If this is an array, the parameter called AlignTitleOrL1 indicates the title alignment.
If this parameter is invalid or numeric, it indicates the title alignment such that if it is true (non-0) the title is included in the calculation for vertical alignment, if false(0) it is added to the droplist after it (and its bevel if one exists) has been vertically aligned. The default is true.

CanEdit

An optional parameter that is any logical expression. If true (non-0) the text displayed in the droplist can be edited in the same manner as an editfield, if false (0) it cannot be edited directly. The default value is false.

Index

An optional parameter that is a variable whose value indicates the array index of the highlighted item in the list. If this information is not required, a constant may be used. This value will be used to calculate the initial value displayed in the droplist (see Init).

FocusID

Boolean. If this value is FALSE (0), the field will display its current setting, but cannot be opened (i.e. its value cannot be changed), and will appear disabled (grayed-out).

Trigger

Optional. If the droplist is editable, Trigger provides feedback. While editing, the value will be 0. When editing is complete (tab, enter or loss of focus) the value will change to non-zero; 1 if enter is pressed, 2 otherwise.

Init

An optional parameter that is any expression for the initial value displayed in the field. The default value is Data[Index]. If Index is invalid, the droplist will initially appear blank.

DrawBevel

An optional parameter that is any logical expression. If true (non-0) a bevel is drawn around the droplist, if false (0) no bevel is drawn. The default value is true.

VertAlign

An optional parameter that is any numeric expression that sets the vertical alignment of the unopened droplist according to one of the following:

VertAlign	Vertical Alignment
0	Top
1	Center
2	Bottom

Whether or not the title is included when the vertical alignment is calculated is determined by the value of AlignTitle. The default value is 0.

AlignTitle_L0

A flexible parameter that is either a logical expression indicating the alignment of the title, or it is the first label (option) in the list that is displayed by this graphic. If the parameter called DataOrAlignTitle is a data array of options, this parameter will indicate the title alignment (see the summary for the DataOrAlignTitle parameter) otherwise this will be any text expression that gives the first label in the dropped list.

RetArray_L1

A flexible parameter that is either the second label in

the list displayed by this graphic, or it can be an array of values to which the parameter at ParmNum will be set. This allows PDropList to be used in places that would otherwise need to use System\Droplist and manually set the value of Paramter[#ParmNum]

L2, L3, ...L16

A list of text expressions giving the remaining options in the dropped list. Any number of these may be left invalid. The resulting list displayed when droplist is opened will contain all valid entries in the 16 available positions.

Comments:

This module is a member of the VTScada Dialog Library and must therefore be called from within a GUITransform and prefaced by \DialogLibrary\.
The "P" tools (PCheckBox, PContributor, PColorSelect, PDropList, and PEditfield) were intended only for use in configuration folders and drawing panel modules, and therefore are subject to the system security restraints.
This parameter tool expects the first parameter of its calling module to contain an array of tag parameters. It will then set the element indicated by ParmNum to the text string displayed in the selected line of the droplist.
The height of the (unopened) droplist is constant, with the horizontal boundaries of its calling transform defining its width, and the vertical boundaries of its calling transform defining its opened height, which will include the added height of the bevel above the field, but may or may not include the title, depending on the alignment used. Note that if the entire list can be displayed in a smaller area

than indicated by the vertical boundaries of the calling transform, the dropped list height will be decreased. The dropped height of the list will always have a minimum height of 1 line (below the field).
For any optional parameter that is to be set, all optional parameters preceding the desired one must be present, although they may be invalid.
Usual height: 40

Example:

The following example illustrates the data options being stored in an array:

```
GUITransform(70, 210, 210, 170,
             1, 1, 1, 1, 1,
             0, 0, 1, 0,
             0, 0, 0,
             \DialogLibrary\PDroplist(6 { Parm num },
             "PLC Type" { Title },
             PLCTypes { Array of data },
             0 { Not editable },
             Idx { Chosen element },
             3 { Focus ID },
             0 { Trigger not req'd },
             Invalid { Use default for init },
             1 { Draw bevel },
             1 { Center list },
             0 { Align top of bevel }));
```

The next example shows how data options may be entered as a grouping of text strings:

```
GUITransform(70, 210, 210, 170,
             1, 1, 1, 1, 1,
             0, 0, 1, 0,
             0, 0, 0,
             \DialogLibrary\PDroplist(6 { Parm num },
             "PLC Type" { Title },
             1 { Align title with top },
             0 { Not editable },
             Idx { Chosen element },
             3 { Focus ID },
             0 { Trigger not req'd },
             Invalid { Use default for init },
             1 { Draw bevel },
             1 { Center list },
```

```
            "PLC-2", "PLC-3", "PLC-5", "PLC-5/250",
            "SLC-500" { Options in list }));
```

PEditfield

(Dialog Library)

Description: Parameter Setting Editfield. This module draws an editfield with (optional) title and bevel.

Returns: Nothing

Usage: Steady State only.

Function Groups: Graphics

Related to: GUITransform | PAddressEntry | PAreaSelect | PCheckBox | PColorSelect | PContributor | PDroplist | PPageSelect | PRadioButtons | PSecBit | PSelectObject | PSpinbox | PTypeToggle

Format: \DialogLibrary\PEditField(ParmNum [,Title, DataType, FocusID, Trigger, View, DrawBevel, VertAlign, AlignTitle, LowLimit, HighLimit, PrivNotReq, Style, PrefixValue, PostfixValue])

Parameters:

ParmNum

Required. Any numeric expression giving the parameter number (from 0) in the caller to alter.

Title

An optional parameter that is any text expression to be used as a title for the field.

DataType

Defaults to 4 unless otherwise specified.

DataType	Type
0	Byte (unsigned)
1	Short (2 byte signed)
2	Long (4 byte signed)
3	Double precision floating point (8 byte signed)
4	Text
5	Octal (4 byte unsigned)
6	Hexadecimal (4 byte unsigned)

For types 0 - 2, if the number entered into the field is prefaced by a "0x" the value is taken to be hexadecimal format, and if it is prefaced by a "0" it is considered to be octal. In either case, the value is converted to decimal format when return is pressed or the focus is lost.

For type 5, regardless of whether or not the number entered into the field is prefaced by a "0" the value is taken to be octal and will be displayed as such. The actual type of Variable will be text.

Type 6, like type 5 will be kept in its declared format of hexadecimal regardless of whether or not the number entered into the field is prefaced by a "0x". The actual type of Variable will be text

FocusID

Boolean. If this value is FALSE (0), the field will display its current setting, but cannot be opened (i.e. its value cannot be changed), and will appear disabled (grayed-

out).

Trigger

Optional. Trigger provides feedback. While editing, the value will be 0. When editing is complete (tab, enter or loss of focus) the value will change to non-zero; 1 if enter is pressed, 2 otherwise.

View

An optional parameter indicating how to display the editfield , as follows. The default value is 2 if FocusID is 0 and 1 otherwise.

View	Display mode
0	Invisible
1	Normal (color scheme - no graying)
2	Grayed-out (only if FocusID is 0)

This parameter may be used to force an editfield with a FocusID of 0 to be displayed normally, rather than allowing it to default to its grayed color.

Note that if the FocusID is not 0, setting this value as 2 will not force the field to gray out.

DrawBevel

An optional parameter that is any logical expression. If true (non-0) a bevel is drawn around the editfield, if false (0) no bevel is drawn. If the editfield is beveled, its size will become fixed and will be the same as that for a droplist. The default value is true.

VertAlign

An optional parameter that is any numeric expression that sets the vertical alignment of the editfield according to one of the following options:

VertAlign	Vertical Alignment
0	Top
1	Center
2	Bottom

Whether or not the title is included when the vertical alignment is calculated is determined by the value of AlignTitle. The default value is 0.

AlignTitle

An optional parameter that is any logical expression. If AlignTitle is true (non-0), the title will be included in the calculation for vertical alignment. If AlignTitle is false (0), the title will be added to the editfield after both the editfield and its bevel have been vertically aligned. The default is true.

LowLimit

An optional parameter that is any expression giving the minimum value or minimum number of characters to be accepted by the editfield (depending on the data type). This value may be a decimal, octal or hexadecimal value. If this parameter is valid and a value less than LowLimit is entered in the field (or there are too few characters, in the case of text value), the variable set by the field will revert to the previous value. No default

HighLimit

An optional parameter that is any numeric expression giving the maximum value or maximum number of characters to be accepted by the editfield (depending on the data type). This value may be a decimal, octal

or hexadecimal value. If this parameter is valid and a value greater than HighLimit is entered in the field (or there are too many characters, in the case of text value), the variable set by the field will revert to the previous value.

PrivNotReq

An optional parameter that is any logical expression. If set to true (non-0), anyone can change the value in this editfield. If set to false (0), only those users whose user accounts have been granted the "TagModify" privilege may set it. The default value is false.

Style

An optional parameter indicating the style of the PEditField object. It is a bit-wise field made up of the sum of the following values, to yield the desired effects.

Bit	Value	Definition
0	1	Reserved for bit compatibility with WinComboCtrl, and should be set to "0"
1	2	Reserved for bit compatibility with WinComboCtrl, and should be set to "0"
2	4	Input is converted to all uppercase (note 1)
3	8	Input is converted to all lowercase (note 1)
4	16	Input is masked. Any characters typed will appear as asterisks. This is useful for such things as password fields
5	32	Multiline editing. Setting this bit causes a typed <CR> (Enter or Return) to be interpreted as "move to the start of the next line". Text that contains <CR> characters has a line break inserted at each
6	64	Reserved
7	128	
8	256	Reserved
9	512	Not used. Height is defined by \EditHt or TEditHt (with title).

If neither values 4 or 8 are set, input is passed to script code as typed.

PrefixValue

An optional parameter indicating the text expression that should be displayed immediately before (i.e. to the left of) the editable part of the control.

SuffixValue

An optional parameter indicating the text expression that should be displayed immediately after (i.e. to the right of) the editable part of the control. No Default

Comments:

This module is a member of the VTScada Dialog Library and must therefore be called from within a GUITransform and prefaced by \DialogLibrary\.
The "P" tools (PCheckBox, PContributor, PColorSelect, PDroplist, and PEditField) were intended only for use in configuration folders and drawing panel modules, and therefore are subject to the system security restraints.
This parameter tool expects the first parameter of its calling module to contain an array of tag parameters. It will then set the element indicated by ParmNum to the value in the editfield.
The height of the editfield is constant, with the horizontal boundaries of its calling transform defining its width, and the vertical boundaries of its calling transform defining the boundaries in which it is to be confined vertically, which will include the added height bevel, but may or may not include the title, depending on the alignment used.
Usual height: 45 pixels.

Examples:

```
GUITransform(10, 215, 225, 15,
             1, 1, 1, 1, 1,
             0, 0, 1, 0,
             0, 0, 0,
             \DialogLibrary\PEditField(1 { Parm num },
             "Manual Data" { Title },
             3 { Type float },
             6 { Focus ID },
             0 { Trigger not req'd },
             Invalid { Use default view },
             0 { No bevel },
             Invalid { Default alignment },
             1 { Align title },
             98.6 { Minimum value },
             Invalid { No maximum value }));
```

Notice in the above example that the last parameter could have been omitted, because it is optional and its setting matches the default.

PEditName

(Dialog Library)

Description: An edit field for setting tag names. This module draws an edit field that is to be connected to the name of a tag. Should be used by all tag ConfigFolder modules for setting the name parameter.

Returns: Nothing

Usage: Steady State only.

Function Groups: Graphics

Related to: GUITransform | PEditfield

Format: \DialogLibrary\PEditName(Trigger[, DrawBevel, Title, ID])

Parameters:

Trigger

Required. Set when the name variable is changed.

DrawBevel

Optional Boolean. When TRUE, a bevel will be drawn.

Defaults to TRUE.

Title

Optional. Any text that you would like display as the title of the edit field. Defaults to "Name".

ID

Optional. Any numeric expression providing the Focus ID of the edit field. Defaults to 1.

Comments: This module is a member of the VTScada Dialog Library and must therefore be called from within a GUITransform and prefaced by \DialogLibrary\.
The "P" tools (PCheckBox, PContributor, PColorSelect, PDroplist, and PEditField) were intended only for use in configuration folders and drawing panel modules, and therefore are subject to the system security restraints.
Usual height: 45 pixels.

Example:

```
{***** Name of the tag *****}
GUITransform(30, 90, 470, 45,
             1, 1, 1, 1, 1  { No scaling                 },
             0, 0, 1, 0     { No movement. Visible. Reserved},
             0, 0, 0        { Not selectable             },
             \DialogLibrary\PEditName(Trigger));
```

PeekStream

Description: Returns a string of bytes from a stream without removing them from the stream.

Returns: See description

Usage: Script only.
May be used in optimized Tag Parameter Expressions.

Function Groups: Stream and Socket

Related to: GetStreamLength | PipeStream | SRead | StreamEnd

Format: PeekStream(Stream, N)

Parameters:

Stream

Required. Any expression returning a pipe stream.

N

Required. Any numeric expression giving the number of bytes to get from the pipe.

Comments: Execution of this function doesn't affect the stream position pointer.
Second parameter must be 65,523 characters or less, otherwise invalid will be returned.

Example:

```
If ! Valid(data);
[
  data = PeekStream(strm, 10);
]
```

This stores the last 10 bytes from strm in data. The stream is unaffected.

Pen

Description: Returns a pen value.

Returns: Pen

Usage: Steady State only.

Function Groups: Color, Graphics

Related to: Brush

Format: Pen(Color, Style, Width)

Parameters:

Color

Required. Any numeric expression giving the color of the line.. Any of the following may be used:

- a palette VTScada Color Palette

- a Constants for System Colors (constant)
- -1 (transparent)
- an RGB string with optional Alpha value in the format, "<AARRGGBB>", or "<RRGGBB>", where AA, RR, GG and BB are hexadecimal digits.

Style

Required. Any numeric expression giving the Line Types. Valid line styles are from 1 to 5 inclusive for standard lines. A line style of 1 is a solid line.
If the Pen command is used within a GUIPolygon to draw a pipe, the use line style 100 (constant: PEN_STYLE_PIPE).

Width

Required. Any numeric expression that gives the line width in pixels.

Comments: Pen values are used in layered graphics statements that draw lines (such as GUIArc or GUIRectangle).

Example:

```
GUIArc(728, 227, 477, 50 { Bounding box for arc },
       1, 1, 1, 1, 1 { No scaling },
       0, 0 { No trajectory or rotation },
       1, 0 { Arc is visible; reserved },
       0, 0, 0 { Cannot be focused/selected },
       Pen(14 { yellow }, 3 { dotted }, 2 { pixel width }),
       Vertex(1 { Double smooth mode },
       Point(602.5, 138.5, Invalid, Invalid),
       Point(710, 293, Invalid, Invalid),
       Point(413, 52, Invalid, Invalid)));
```

This shows how a Pen statement may be used to affect the attributes of a drawing object. In the above example, the arc will be drawn with a dotted yellow line 2 pixels wide.

Pending

Description: Returns the number of statements of a certain type pending.

Returns: Numeric

Usage: Script Only.

Function Groups: Graphics

Related to: Priority | PriorityWeight

Format: Pending(StatementType)

Parameters:

StatementType

Required. Any numeric expression giving the type of statement for which to get the number pending as follows:

StatementType	Pending type
0	High priority statement
1	Normal priority statement
2	Timer
3	Priority statement
4	Logger queue length (logQsize in LogFile.cpp)

Example:

```
If 1 Next;
[
  numPending = Pending(0);
]
This script finds the number of high priority statements pending at
the time of execution and assigns the value to the variable numPend-
ing.
```

PersistentSize

Description: Returns the size in bytes of a variable's persistent value size in the persistent value (.VAL) file.

Warning: This function should be used by advanced users only.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Compilation and On-Line Modifications, Variable

Related to: AddVariable | ChangePersistentSize | FindVariable | MakeNonPersistent | MakePersistent

Format: PersistentSize(Variable)

Parameters:

Variable

Required. Any expression for the variable value. This value is usually returned from a call to AddVariable or FindVariable.

Comments: If Variable is not a persistent variable the function will return invalid.

PHSliderBar

(Dialog Library)

Description: Called graphic module that connects a horizontal slider bar to a given parameter number.

Returns: Nothing

Usage: Steady State only.

Function Groups: Graphics

Related to: GUITransform | PAddressEntry | PAreaSelect | PCheckBox | PContributor | PDroplist | PEditfield | PPageSelect | PRadioButtons | PSecBit | PSelectObject | PSpinbox | PTypeToggle

Format: \DialogLibrary\PHSliderBar(ParmNum [, Title, FocusID, Trigger, DrawBevel, MinValue, MaxValue, EnableEditField])

Parameters:

ParmNum

Required. Any numeric expression giving the parameter number (from 0) in the caller to alter.

Title

An optional parameter that is any text expression to be used as a the title to put on the bevel.

FocusID

Boolean. If this value is FALSE (0), the field will display its current setting, but cannot be opened (i.e. its value cannot be changed), and will appear disabled (grayed-out).

Trigger

Set when the variable is changed.

DrawBevel

Any Boolean expression which, when set to TRUE, indicates that a bevel is to be drawn around the control.

MinValue

Any numeric expression for the minimum of Value's range.

MaxValue

Any numeric expression for the maximum of Value's range.

EnableEditField

An optional parameter that is any logical expression. If true (non-0) the edit field will be shown. The default is FALSE.

Comments: This module is a member of the VTScada Dialog Library and must therefore be called from within a GUITransform and prefaced by \DialogLibrary\.
The "P" tools (PCheckBox, PContributor, PCo-

lorSelect, PDroplist, and PEditfield) were intended only for use in configuration folders and drawing panel modules, and therefore are subject to the system security restraints.
This parameter tool expects the first parameter of its calling module to contain an array of tag parameters. It will then set the value of the element indicated by ParmNum.
Usual height: 40 pixels.

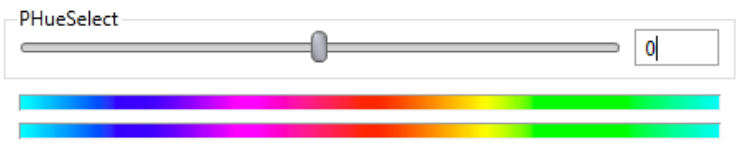
Example:

```
  GUITransform(70, 200, 290, 160,
               1, 1, 1, 1, 1          { No scaling
},
               0, 0, 1, 0             { No movement; visible;
reserved },
               0, 0, 0                { Not selectable
},
               \DialogLibrary\PHSliderBar(#RelativeSize,
                                          "Set Size", { Title
},
                                          5         { ID
},
                                          Trigger   { Trigger
},
                                          0         { No Bevel
},
                                          0         { Min scale value
},
                                          100       { Max scale value
},
                                          1         { Enable edit
}));
```

PHueSelect

(Dialog Library)

Description: Called graphic module that connects a hue selection tool to a given parameter number.



Returns: Nothing

Usage: Steady State only.

Function Groups: Color, Graphics

Related to: GUITransform | PAddressEntry | PAreaSelect | PCheckBox | PColorEdit | PContributor | PDroplist | PEditfield | PPageSelect | PRadioButtons | PSecBit | PSelectObject | PSpinbox | PTypeToggle

Format: \DialogLibrary\PHueSelect(ParmNum [, Title, FocusID, Trigger, DrawBevel, EnableEditField])

Parameters:

ParmNum

Required. Any numeric expression giving the parameter number (from 0) in the caller to alter.

Title

An optional parameter that is any text expression to be used as a the title to put on the bevel.

FocusID

Boolean. If this value is FALSE (0), the field will display its current setting, but cannot be opened (i.e. its value cannot be changed), and will appear disabled (grayed-out).

Trigger

Set when the variable is changed.

DrawBevel

Any Boolean expression which, when set to TRUE, indicates that a bevel is to be drawn around the control.

EnableEditField

An optional parameter that is any logical expression. If true (non-0) the edit field will be shown. The default is FALSE.

Comments: This module is a member of the VTScada Dialog Library and must therefore be called from within a GUITransform and prefaced by \DialogLibrary\.
The "P" tools (PCheckBox, PContributor, PColorSelect, PDroplist, and PEditfield) were intended only for use in configuration folders and drawing panel modules, and therefore are subject to the system security restraints.
This parameter tool expects the first parameter of its calling module to contain an array of tag parameters. It will then set the value of the element indicated by ParmNum.
Usual height: 77 pixels.

Example:

```
 GUITransform(30, 200, 300, 270,
              1, 1, 1, 1, 1            { no scaling
},
              0, 0, 1, 0               { No movement; visible;
reserved },
              0, 0, 0                  { Not selectable
},
              \DialogLibrary\PHueSelect(#MismatchHue,
                                        Invalid   { Title },
                                        10        { ID    },
                                        Trigger,
                                        0, 1));
```

Pick

Description: Returns an indication of whether the locator (e.g. mouse) has had a specified change in its button status.

Returns: Numeric

Usage: Steady State only.

Function Groups: Graphics, Locator

Related to: Click | LocSwitch | SetXLoc | SetYLoc | Target | WinLocSwitch | WinXLoc | WinYLoc | XLoc| YLoc |

GUITransform

Format: Pick(X1, Y1, X2, Y2, Button)

Parameters:

X1

Required. Any numeric expression giving the X coordinate on the screen of one side of the screen area ("target").

Y1

Required. Any numeric expression giving the Y coordinate on the screen of either the top or bottom of the screen area ("target").

X2

Required. Any numeric expression giving the X coordinate on the screen of the side of the "target" opposite to X1.

Y2

Required. Any numeric expression giving the Y coordinate on the screen of either the top or bottom of the target, whichever is the opposite of Y1.

Button

Required. Any numeric expression giving the button combination that activates this graphic when the locator cursor is within the "target" screen area.

Button	Button Locator
0	No buttons
1	Right button
2	Middle button
3	Right and middle buttons
4	Left button
5	Left and right buttons
6	Left and middle buttons
7	All three buttons

If the value is:
- multiplied by 8 the meaning for multiple buttons pressed becomes "OR" rather than "AND." For example, to accept any button on a 2 or 3 button mouse, use 56 (8 * 7). To accept the left mouse button regardless of whether or not the right button is pressed, use 32 (8 * 4)
- increased by 64, the function will become true when the mouse buttons are released rather than when they are pressed
- increased by 128, the button(s) must be double-clicked
- increased by 256, forces the control with the input focus (in the same window as the Pick) to simulate the Enter key being pressed prior to the Pick function returning a value of true. In other words, with code similar to:

```
If Pick(X1, Y1, X2, Y2, 0X104);
[
…script statements…
]
```

any value affected by the control with input focus will be set to the current value of the control prior to the script statements being run.
- increased by 512, horizontal panning is disabled over the region.
- increased by 1024, vertical panning is disabled over the region.
These last two options, 512 and 1024, allow Pick, WinLocSwitch and other events to be handled by a region within the window - for example a map within a Sites page when viewed using the VTSCada Anywhere Client. These should be used only by advanced users.

Comments: This function returns true if the locator button combination changes as specified by the Button parameter while the locator position is within the boundaries of the "target" ((X1,Y1) - (X2,Y2)). This function is "edge triggered" which means that it only acts upon the changes in locator button status. Making the button combination specified by the Button parameter outside the target area and then sliding the cursor into the target will not cause the function to return true. If the locator is not installed, the function will return false (0).
This function is very useful for capturing fast mouse button presses by the operator. VTScada remembers the locations where the mouse buttons were pressed and released. The actual buttons do not have to be pressed when the Pick is actually executed since it will examine the list of all button

changes since the last time it was executed.
This function is latched on once it becomes true.
Use Click() for steady state feedback rather than Pick().
Note: This function is disabled when using a GUITransform as a GUIStrectch.

Example:

```
If Pick(300, 500, 600, 700, 56) MixerScreen;
```

This action will switch to the MixerScreen state when the mouse is inside the target box and any mouse button is pressed.

Related Information:

See: "Latching and Resetting Functions" in the VTScada Programmer's Guide

PickValid

Description: Attempts to return a valid value given a list of parameters.

Returns: Varies

Usage: Script or steady state.

Function Groups: Logic Control

Related to: Valid

Format: PickValid(Parm1, Parm2 [, Parm3, ...])

Parameters:

Parm1, Parm2, Parm3...

Required. Any number of parameters giving any expressions, from which the first valid value will be selected.

Comments: This function continues its search through its parameter list in order until the first valid value is found.

Examples:

```
a = Invalid;
b = 83;
c = Invalid;
d = -1;
validAns1 = PickValid(a, b, c, d);
validAns2 = PickValid(a, c, d, b);
```

The value of validAns1 and validAns2 will be 83 and -1 respectively. A typical use for this function is in setting default values as follows:

```
<
MyEditfield
(
  left;
)
SetDefaults
[
  If 1 Main;
  [
   left = PickValid(left, 100);
  ]
]
...
>
```

Given that module MyBox draws a box in the window, adding the Pick-Valid statement ensures that even if an invalid value is given as a parameter, the box will still be displayed, because its invalid parameter will have been replaced with the default value.

PID

Description	Perform PID Controller Function. This function returns a control value to maintain a parameter at a given setpoint.
Returns	Numeric
Usage	Steady State only. See: Rules for Usage.
Function Groups	Variable
Related to:	Deriv | Intgr
Format:	PID(PV, SP, Mode, Track, LowLimit, HighLimit, P, I, D, Bias, Time)
Parameters	

PV

Required. Any numeric parameter which gives the "process variable" to be maintained at the given setpoint.

SP

Required. Any numeric parameter that gives the value to use for the "setpoint." The PID function will change its return value to make the PV value match the SP value.

Mode

Required. Any numeric parameter giving the manual/auto mode for the function. When the parameter is not equal to 0, the function is in auto mode and uses the PID algorithm to control the output.
When the parameter is equal to 0, the function simply returns the value of the Track parameter. When the mode switches from manual to auto, the PID function implements a "bumpless transfer." This results in a smooth output change rather than an abrupt change. This is done by forcing the internal integral to assume a value which forces the PID function to output a value which matches the Track parameter at the moment of the change. The integral value will not, however,

Mode	Algorithm	Derivative	Gains
0	Manual	N/A	N/A
1	Normal	Uses PV	Independent
2	Normal	Uses PV	Dependent
3	Normal	Uses Error	Independent
4	Reverse	Uses Error	Dependent
5	Reverse	Uses PV	Independent
6	Reverse	Uses PV	Dependent
7	Reverse	Uses Error	Independent

The PID function operates in eight automatic modes and one manual mode. These modes are listed in the above table. If in doubt, use mode 1 or 5.

The Action column in the above table indicates whether the PID function operates using normal action (SP - PV) or reverse action (PV - SP). With normal action, when PV increases, the function output decreases. With reverse action, when PV increases, the function output increases also.

The Derivative column in the above table indicates whether the PID function derivative bases its value upon PV or the error between PV and SP. For normal action, the error is (SP - PV). For reverse action, the error is (PV - SP). The difference between the two modes is only clear when the setpoint (SP) is changed. If the derivative uses PV, the response to setpoint changes is normal. If the derivative uses the error, the response to setpoint changes is faster but usually produces an abrupt change in the PID function output which is often undesirable.

The Gains column in the above table indicates which PID equation is used. Refer to the "Comments" section for these equations. The independent gains equation uses seconds as the time base and the gain parameter (P) does not affect either the integral or derivative. The dependent gain equation is the ISA equation. It uses minutes as the time base and the gain parameter (P) affects both the integral and derivative.

Track

Required. Any numeric parameter that gives the function output value when in manual mode (i.e. Mode equal to 0). When in auto mode, this parameter is ignored except at the instant immediately following the change from manual to auto mode.

LowLimit

Required. Any numeric expression giving the minimum value allowed for the PID function. The integral component may grow to a very large value if the setpoint (SP) is outside the controllable range.

Physical plant limitations may prevent the PV value from reaching the SP value. In such situations, the controller will respond very slowly to bringing the SP within the controllable range since it will take a long time for the large integral value to increase to the correct value.

The LowLimit parameter puts a lower bound on the value of the PID function. It does this by limiting the value of the internal integral that results in instantly regaining control once the setpoint is brought back into the controllable range. This feature is sometimes called "anti-reset windup."

HighLimit

Required. Any numeric expression giving the maximum value allowed for the PID function. This is similar to the LowLimit parameter except that it handles the upper limit for the PID function output.

P

Required. Any numeric expression giving the "proportional" or "gain" contribution to the PID output. It has no units (dimensionless).

I

Required. Any numeric expression giving the "integral" or "reset" contribution to the PID output. For independent gains, the units are inverse seconds and the I parameter is used directly as the gain for the integral portion of the PID equation.

For dependent gains, the units are minutes and the P parameter divided by I is used as the gain for the integral portion of the PID equation. Refer to the "Comments" section for the equations.

D

Required. Any numeric expression giving the "derivative" or "rate" contribution to the PID output. For independent gains, the units are seconds and the D parameter is used directly as the gain for the derivative portion of the PID equation.

For dependent gains, the units are minutes and P times D is used as the gain for the derivative portion of the PID equation. Refer to the "Comments" section for the equations.

Bias

Required. Any numeric expression giving the output offset or feed forward input. This value is added to the output value. Refer to the equations in the "Comments" section.

Time

Required. Any numeric expression giving the time interval in seconds between integral and derivative updates for the PID function. This value results in a smoothing of the derivative values for the sampled process data of VTScada .

The longer the time, the greater the smoothing. The shorter the time, the faster the controller response. A suggested time interval is the I/O update time for the PV parameter.

Comments

Typically, this function is used to set a variable that controls a process output point. The equations used for this function follow.

Independent Gains Equation:

```
CV = P * E + I *∫ E dt + D * dA/dt + Bias
```

Dependent Gains Equation (ISA):

```
CV = P * (E + 1/I *∫ E dt + D * dA/dt) + Bias
```

Where:

- *CV* PID function output.
- *E* Error. E = (SP – PV) for normal action modes and E = (PV – SP) for reverse action modes.
- *A* Derivative selection. For modes with the derivative based upon the PV value, A = PV. For modes with the derivative based upon the error value, A = E.

PID control may be done by a PLC or other device rather than VTScada. Setpoints, coefficients, and so forth may be read and written by VTScada I/O drivers. This form of PID control does not involve use of the VTScada PID function. See the reference manual of the PLC or other device and the VTScada I/O driver manual for more.

Example:

```
{ Valve loop controller for heat exchanger }
valvePos = PID(cooledTemp { Process value },
               cooledSP { Setpoint for cooled temperature },
               Cond(valveManual, 0, 5)
               { Manual or reverse acting deriv
               uses PV, independent gains },
               manValvePos { Track value used in manual mode },
               0, 1 { Lower/upper limits of valve position },
               0.1 { P coefficient - proportional gain },
               0.5 { I coeff - integral gain or reset time },
               0.02 { D coefficient - derivative gain or rate },
               0 { Output bias },
               0.5 { Update loop every half second });
```

This performs PID control of a heat exchanger. A valve is used to control a temperature given a setpoint. If valveManual is true, the PID simply updates its internal integrators every half second using manValvePos. If valveManual is false, PID control is updated every half second. The P, I, and D gains are constants, but might be variables, to allow online tuning.

Pie

Note: Deprecated. Do not use in new code.

Description Draws a pie shaped wedge on the screen.

Returns Nothing

Usage Steady State only.

Function Groups Graphics

Related to: Arc | Ball | Circle | Ellipse | GUIArc | GUIChord | GUIEllipse | GUIPie

Format: Pie(X, Y, Radius, Angle1, Angle2, Foreground, Pattern, Background)

Parameters

X

Required. Any numeric expression giving the X coordinate of the center of the pie on the screen.

Y

Required. Any numeric expression giving the Y coordinate of the center of the pie on the screen.

Radius

Required. Any numeric expression giving the radius of the pie specified in units of X screen coordinates.

Angle1

Required. Any numeric expression giving the starting angle of the pie in radians. An angle of 0 lies on the X axis to the right of the center of the pie.

Angle2

Required. Any numeric expression giving the ending angle of the pie in radians.

Foreground

Required. Any numeric expression giving the color index for the foreground color of the pie fill pattern.

Pattern

Required. Any numeric expression giving the hatch Fill Patterns to use for the fill. The valid hatch style numbers are from 1 to 25 inclusive.

Background

Required. Any numeric expression giving the background color for the hatch pattern. This value is only significant if the Pattern parameter is not equal to 1.

Comments

This statement has been superseded by the GUIPie function and is maintained for backwards compatibility only.
The pie is drawn in a counterclockwise direction from the Angle1 to Angle2.
As of version 11, this is now drawn in the same z-order as other graphics, making it similar to the z-graphics functions.

Example:

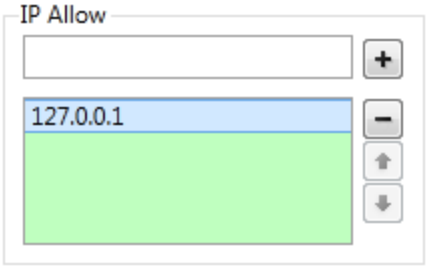
```
Pie(100,200 { X-Y coordinates for the center of the pie },
    200 { Radius in pixels },
    0 { Start angle (3 o'clock position) },
    3.14 { Draw counterclockwise for 180 degrees },
    10, 1, 0 { Light green solid (background ignored });
```

This draws the top half of a light green circle.

PIPAddressList

(Dialog Library)

Description: Uses an IPAddressList to set a parameter with a semicolon-delimited IP address list.



Returns: Nothing

Usage: Steady State only.

Function Groups: Graphics

Related to: GUITransform | IPAddressList | PAddressEntry | PAreaSelect | PCheckBox | PColorEdit | PContributor | PDroplist | PEditfield | PPageSelect | PRadioButtons | PSecBit | PSelectObject | PSpinbox | PTypeToggle

Format: \DialogLibrary\PIPAddressList(ParmNum [, Trigger, FocusID, Title, DrawBevel, AlignTitle])

Parameters:

ParmNum

Required. Any numeric expression giving the parameter number (from 0) in the caller to alter.

Trigger

Set when the variable is changed.

FocusID

Boolean. If this value is FALSE (0), the field will display its current setting, but cannot be opened (i.e. its value cannot be changed), and will appear disabled (grayed-out).

Title

An optional parameter that is any text expression to be used as a the title to put on the bevel.

DrawBevel

Any Boolean expression which, when set to TRUE, indicates that a bevel is to be drawn around the control.

AlignTitle

An optional parameter that is any logical expression. If true (non-0) the title is included in the calculation for vertical alignment, if false(0) it is added to the droplist after it and its bevel has been vertically aligned. The default is true. Usual height: 130 pixels.

Comments:

This module should be used whenever the user needs an IP Allow filter for use with the SocketManagerServer.

The module is a member of the VTScada Dialog Library and must therefore be called from within a GUITransform and prefaced by \DialogLibrary\.
The "P" tools (PCheckBox, PContributor, PColorSelect, PDroplist, and PEditfield) are intended only for use in configuration folders, and therefore are subject to the system security restraints.
This parameter tool expects the first parameter of its calling module to contain an array of tag parameters. It will then set the value of the element indicated by ParmNum.
The recommended minimum height ranges from 100 pixels for a plain list to 130 pixels for a list with an aligned titled bevel. The up/down buttons will hide if there is not enough room for them to be displayed.

Example:

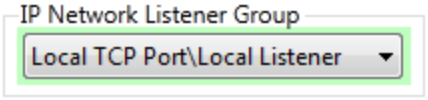
```
{***** Allowed incoming IP address (or ranges) -- optional *****}
GUITransform(30, 309, 240, 177, { Ht 130 }
             1, 1, 1, 1,
             1, 0, 0, 1, 0, 0, 0, 0,
             \DialogLibrary\PIPAddressList(\#IPAddressAllow { ParmNum
},
               Trigger,
           Valid(Parms[\#ListenerGroup]) ?  10 { FID to 19} : 0,
```

```
            \IPAllowLabel { Title},
                TRUE { DrawBevel },
               TRUE { AlignTitle }));
```

PIPListenerGroup

Dialog Library

Description: Draws a droplist of all available IP Listener Groups. (In general, the groups will be the IP Listener tag names.)



Returns: Nothing

Usage: Steady State only.

Function Groups: Graphics

Related to: GUITransform | PAddressEntry | PAreaSelect | PCheckBox | PContributor | PDroplist | PEditfield | PPageSelect | PRadioButtons | PSecBit | PSelectObject | PSpinbox | PTypeToggle

Format: \DialogLibrary\PIPListenerGroup(ParmNum, Trigger, FocusID [, Title, DrawBevel, VertAlign, AlignTitle])

Parameters:

ParmNum

Required. Any numeric expression giving the parameter number (from 0) in the caller to alter

Trigger

Set when the variable is changed.

FocusID

Boolean. If this value is FALSE (0), the field will display its current setting, but cannot be opened (i.e. its value cannot be changed), and will appear disabled (grayed-out).

Title

An optional parameter that is any text expression to be used as a the title to put on the bevel.

DrawBevel

Any Boolean expression that, when set to TRUE, indicates that a bevel is to be drawn around the control.

VertAlign

An optional parameter that is any numeric expression that sets the vertical alignment of the droplist according to one of the following options:

VertAlign	Vertical Alignment
0	Top
1	Center
2	Bottom

Whether or not the title is included when the vertical alignment is calculated is determined by the value of AlignTitle. The default value is 0.

AlignTitle

An optional parameter that is any logical expression. If true (non-0) the title is included in the calculation for vertical alignment, if false(0) it is added to the droplist after it and its bevel has been vertically aligned. The default is true.

Comments:

This module is a member of the VTScada Dialog Library and must therefore be called from within a GUITransform and prefaced by \DialogLibrary\.
The "P" tools are intended for use only in configuration folders and drawing panel modules, and therefore are subject to the system security restraints.

This parameter tool expects the first parameter of its calling module to contain an array of tag parameters. It will then set the value of the element indicated by ParmNum.
Usual height: 45 pixels.

Examples:

```
{***** Network Listener group *****}
GUITransform(30, 168, 240, 123, { Ht 45 }
             1, 1, 1, 1, 1 { No scaling },
             0, 0, 1, 0 { No movement; visible; reserved },
             0, 0, 0 { Not selectable },
             \DialogLibrary\PIPListenerGroup(\#ListenerGroup, Trig-
ger,9 {ID},
             \IPNetworkListenerGroup));
```

Pipe

Note: Deprecated. Do not use in new code.

Description: Draw a double line

Returns: Nothing

Usage: Steady State only.

Function Groups: Graphics

Related to: GUIPipe | Line | ZPipe

Format: Pipe(Width, Color, Curvature, X1, Y1, X2, Y2, ...)

Parameters:

Width

Required. Any numeric expression giving the spacing of the lines in units of X screen coordinates. The width is always rounded to result in an odd number of pixels on the screen. The minimum width displayed will be 1 pixel.

Color

Required. Any numeric expression giving the VTScada

Color Palette of the pipe.

Curvature

Required. Any numeric expression giving the radius of curvature of the corners for the pipe. This is specified in units of X screen coordinates.
If the number of endpoints is 2, the Curvature is ignored.

X1, Y1, X2, Y2, ...

Required. Any numeric expressions giving the screen coordinates of the pipe endpoints.

Comments: The radius of curvature of the pipe corners is the radius of the arc that joins the line endpoints. A Curvature of 0 results in sharp (square) pipe corners. Larger Curvature numbers result in greater rounding of the pipe corners.
This statement is useful for drawing piping on the screen, especially if it changes dynamically.

Note: Within an Anywhere Client session, this function does nothing.

Example:

```
Pipe(10 { Width of pipe in pixels },
     11 { Light cyan color },
     20 { Radius of curvature in pixels },
     100, 100 { Coordinates of first point },
     0, 900 { Coordinates of second point },
     100, 200 { Coordinates of third point },
     200, 500 { Coordinates of fourth point });
```

This statement draws a cyan colored pipe, 50 pixels wide between four points.

PipeStream

Description: Returns a stream based on an operating system named pipe.

Returns: Stream

Usage: Script Only.

Function Groups: Stream and Socket

Related to: BlockWrite| BuffStream | CloseStream | FileStream | GetStreamLength | PeekStream | SRead | StreamEnd | SWrite

Format: PipeStream(Name, Mode)

Parameters:

Name

Required. Any text expression giving the name of the named pipe.

Mode

Required. Any numeric expression giving the manner in which the pipe is opened, as shown in the following table:

Mode	Open state
0	Not opened yet
1	Opened for read
2	Opened for read or write

Comments: None

Example:

```
If ! Valid(pStream);
[
  pStream = PipeStream("\\dataserv\datapipe", 1);
]
```

This opens a pipe named pStream as a stream from which to read data.

PixelColor

Description: Returns the color of a pixel in the window.

Returns: Numeric or String (see 4th parameter)

Usage: Script or steady state.

Function Groups: Color, Graphics

Related to: GetSystemColor

Format: PixelColor(Object, X,Y[,UseRGB])

Parameters:

Object

Required. Any expression that returns an object value. This identifies the window where the pixel is drawn.

X

Required. Any numeric expression, giving the x-axis coordinate of the location of the pixel.

Y

Required. Any numeric expression, giving the y-axis coordinate of the location of the pixel.

UseRGB

Optional. Any Boolean expression, which when true, will cause the pixel color to be returned as an RGB string. If false, the numeric palette value will be returned.

Comments: Location (x, y) is taken to be in the window where the module instance identified by Object is drawn.

Example:

```
mouseColor = PixelColor(Self(),XLoc(),YLoc());
```

This finds the color under the mouse in the window where this function is defined.

Platform

Descrip- Returns a twelve element structure that indicates the platform under

tion:	which VTScada is currently running.
Returns:	Structure – see comments
Usage:	Script only. May be used in optimized Tag Parameter Expressions.
Function Groups:	Network, Software and Hardware
Related to:	WKStaInfo
Format:	Platform()
Para-meters:	None

Com-
ments:

Element	Value
PlatformId	0 = Windows ME, Windows 98 SE, Windows 95 1 = Windows NT, Windows 2000, Windows XP, Windows Server 2003, Windows Server 2008, Windows Vista, Windows 7
MajorVersion	Major version number
MinorVersion	Minor version number
ProductType	Additional information about the system.
SuiteMask	Identifies the product suites available on the system. For a list of the codes and their meanings, please refer to http://msdn2.microsoft.com/en-us/library/ms724833.aspx
BuildNumber	The build number of the operating system
SPName	Service Pack Name
SPMajorVersion	Major version # of latest service pack installed
SPMinorVersion	Minor version # of latest service pack installed
ProductInfo	For a list of the codes and their meanings, please refer to http://msdn2.microsoft.com/en-us/library/ms724358.aspx
CPUArchitecture	A 0 indicates x86 architecture, while a 9 indicates x64 architecture
NumberOfCPUs	The number of processors in the machine.

Example:

```
If 1 Main;
[
  p = Platform();
    IfThen(p\MajorVersion == 6 && p\MinorVersion == 0
 && p\ProductType == 1,
      UsingVista = 1;
    );
];
```

This script checks to see if the platform is Windows Vista

Play

Description: Plays a multimedia sound file as installed in the operating system. It differs from Sound in that it is a steady-state statement and is supported by VTScada Internet Client.

Returns: Nothing

Usage: Steady State only. See: Rules for Usage.

Function Groups: Speech and Sound

Related to: Sound

Format: Play(File, Option, Enable [, DevID])

Parameters:

File

Required. Any text expression giving the file name to play. If the extension is omitted, the default extension ".WAV" is added. If an empty string is provided here, then any currently playing sound is stopped.

Option

Required. Any numeric expression that indicates how to play the file. The value of Option may be obtained by adding together numbers from the following table:

Value	Bit No.	Option
1	0	Play asynchronously (don't wait)
2	1	Don't use default sound if file missing.
4	2	Reserved for future use.
8	3	Loop the sound until next Sound function executed.
16	4	Don't stop any currently playing sound.

If Option is "0", VTScada will halt all execution until the sound is finished. This is not recommended. Add "1" to avoid this behavior.

Enable

Required. A value indicating whether or not the sound is played. This can be one of:

Value	Description
1	Sound is played.
0 or Invalid	The sound being played is stopped.

DevID

An optional parameter that is required when you wish to play a sound through a device other than the default system audio device. The ModemDev function can return the identifier of the wave device for a voice modem.

Option value 8 is the only option considered when

DevID has been set.
Note: DevID is ignored for VIC sessions. Options must be configured appropriately if there is a chance that playback might occur over a VTScada Internet Client.

Comments: The return value is a Boolean flag that indicates whether the file has begun playback. There is no return value to signal when playback has finished. Playback will stop when any of the following conditions occur:

- Enable is set to false
- File is set to Invalid or ""
- The steady state call, Play() is stopped.

If the session is remote via a VTScada Internet Client, the return value of this function will not be set to true until the audio file has been completely transferred and the playback has begun remotely.

Plot

Description: Displays a plot of a subsection of a numeric array in a particular area of the window.

Returns: Nothing

Usage: Steady State only.

Function Groups: Graphics

Related to: PlotBuff | PlotXY

Format:

Format version 1:
Plot(ArrayElem, N, Low, High, X1, Y1, X2, Y2, Style, Color, Pattern, PatternBackground)

Format version 2:
Plot(ArrayElem, N, Low, High, X1, Y1, X2, Y2, Pen, Brush [, XORMode, Direction, DrawStepped, BitNumber, Average])

Note: Plot takes the first parameter set or else detects that the Style parameter is a Pen, in which case the second parameter set is used.

Parameters:

ArrayElem

Required. Any numeric expression specifying the starting array element to plot. If processing a multidimensional array, the usual rules apply to decide which dimension should be used.

N

Required. Any numeric expression giving the number of array elements to plot starting at the element given by the first parameter. N must be greater than or equal to 2, and no greater than 32000.
If this parameter is greater than the dimension of the first array, the number of points plotted will be the array dimension.
If N extends past the upper bound of the lowest array dimension, this computation will "wrap-around" and resume at element 0, until N elements have been processed.

Low

Required. Any numeric expression that defines the limit of data values to be plotted. This value need not be less than High but must not be equal to it.
In the standard case, when this value is less than High, it will define the minimum value to be displayed. For example, if Low was 10, a data value of 10 would fall on the edge of the bounding box, on either the line described by X1, if the plot is vertical, or on the line described by Y1, if the plot is horizontal. Any values below 10 in this case would not be shown since they would be outside the clipped box.

High

Required. Any numeric expression that defines the limit of data values to be plotted. This value need not be greater than Low but must not be equal to it. In the standard case, when this value is greater than Low, it will define the maximum value to be displayed. For example, if High was 1000, a data value of 1000 would fall on the edge of the bounding box, on either the line described by X2, if the plot is vertical, or on the line described by Y2, if the plot is horizontal. Any values above 1000 in this case would not be shown since they would be outside the clipped box.

X1

Required. Any numeric expression giving the X coordinate on the screen of one side of the plot area. This is typically, although not necessarily, the left side of the plot.

For horizontal plots, this is the screen coordinate of the first point or bar plotted. For vertical plots, this is the value which corresponds to the Low parameter value. For vertical bar plots, X1 is the base coordinate for the bars.

Y1

Required. Any numeric expression giving the Y coordinate on the screen of one side of the plot area. This is typically, although not necessarily, the bottom of the plot.

For vertical plots, this is the screen coordinate of the first point or bar plotted. For horizontal plots, this is the value which corresponds to the Low parameter value. For horizontal bar plots, Y1 is the base coordinate for the bars.

X2

Required. Any numeric expression giving the X

coordinate on the screen of the side of the plot area opposite to X1. This is typically, although not necessarily, the right side of the plot.
For horizontal plots, this is the screen coordinate of the last point or bar plotted. For vertical plots, this is the value which corresponds to the High parameter value.

Y2

Required. Any numeric expression giving the Y coordinate on the screen of the side of the plot area opposite to Y1. This is typically, although not necessarily, the top of the plot.
For vertical plots, this is the screen coordinate of the last point or bar plotted. For horizontal plots, this is the value which corresponds to the High parameter value.

Style

Required (Format version 1 only). Any numeric expression giving the style of the plot. The plot Style gives the line style, direction, bit number for bits plots, and the number of points to average per point displayed.

The Style is determined by adding together the following values:
Line style + Direction + Path + Bit number + Average

Line style is a number between 1 and 5 inclusive which gives the line style for line plots. A line style of 1 indicates a solid line (see Chapter 9 for more styles). For bar plots, the line style portion of the Style parameter is ignored.

Direction is either 0 or 10. Use 0 for a standard hori-

zontal plot (array index runs along the X axis and values are plotted up or down). Use 10 for a vertical plot (array index runs along the Y axis and values are plotted left or right).

Path is either 0 or 50. A 0 indicates that line plots are to be drawn directly from point to point. A 50 indicates that plots are to be drawn in a step fashion with two line segments between each point. The first segment runs parallel to the X-axis and the second segment runs parallel to the Y-axis. This produces a square-looking plot. These step plots are useful for plotting status values which change in jumps rather than continuously. The Path value is ignored for bar plots.

Bit number is either 0 or a value starting at 100. This is the number of the bit to use in the array data for plotting. If the normal entire value of the data is to be plotted, use 0 for this value. If only one of the bits from the data is to be plotted, use the bit number plus 1. The value to add is (Bit number + 1) * 100 or 0. The ability to plot only a single bit from an array of short or long values allows status data to be stored very efficiently by not requiring a separate array for each status value. For bit plots, the value plotted is always a 0 or a 1 corresponding to the value of the selected bit. The use of this plotting option is usually done in conjunction with a Path value of 50 for step plots.

Average is either 0 or a value starting at 10000. This is the number of consecutive array points to average to give a single point on the screen. It can be used to plot a very large array of values on the screen without having to draw all of the points on the screen. This will produce a smoothed plot and reduce the drawing time.

The number to add is (Data points per displayed point - 1) * 10000.

Color

Required (Format version 1 only). Any numeric expression giving the VTScada Color Palette of the line and the foreground color for filled plots. If the number is less than 10000, the plot is non-destructive. If the number is greater than or equal to 10000, the plot is destructive and the actual color used is Color - 10000. RGB color strings may not be used.

Pattern

Required (Format version 1 only). Any numeric expression giving the bar Fill Patterns for the plot. For a Pattern value of 0, the plot is a line plot.
For Pattern values in the range of 1 to 25 inclusive, the plot is a bar plot with the hatch pattern corresponding to the Pattern parameter value. If the parameter is 1, the bars are a solid color and the Background parameter is ignored.

PatternBackground

Required (Format version 1 only). Any numeric expression giving the color number of the plot fill background color. This number is ignored if the Pattern parameter is equal to 1. For values of Pattern greater than one, it gives the background color for the bar.
RGB color strings may not be used.

Pen

Required (Format version 2 only). The pen to be used (if any). Must be provided, but will be ignored for bar plots.

Brush

Required (Format version 2 only). The brush to be

used (if any). For line drawing, this parameter must be provided with pattern 0.

XORMode

(Format version 2 only) An optional parameter specifying whether or not the plot is destructive (if destructive, the plot line won't be affected by the background color).
If set to 0 or Invalid (default), the destructive drawing mode will be used. If set to 1, the non-destructive or XOR drawing mode will be used.

Direction

An optional parameter specifying the horizontal or vertical direction for the plot. 0 or Invalid indicates horizontal, while 1 indicates vertical.

DrawStepped

(Format version 2 only) An optional parameter specifying whether the plot is to be drawn from point-to-point, or stepped. 0 or Invalid indicates point-to-point, while 1 indicates stepped.

BitNumber

(Format version 2 only) An optional parameter specifying the use of a whole value or a bit number. Invalid indicates a whole value should be used, while >=0 indicates that the specified bit number should be used. No default

Average

(Format version 2 only) An optional parameter specifying whether or not consecutive points should be averaged to give a single point. Invalid or 0 indicates that averaging should not be used, while >0 indicates that the number of consecutive points should be averaged to give a single point. No default.

Comments:

This function is a layered graphics statement.
The Plot statement will execute timer functions during the plotting process. This will allow time critical functions such as driver I/O and data logging to continue running during the relatively long times it takes to update a plot on the screen.

Through the ordering of the Low, High, X1, X2, Y1, and Y2 parameters, the plot may have 8 different orientations for line plots and 16 different orientations for bar plots. The normal left to right plot with the minimum at the bottom will have Low < High, X1 < X2, and Y1 < Y2.

By exchanging Low and High (i.e. Low > High), the plot will have the minimum value at the top with bars still drawn from the bottom. Exchanging X1 and X2 (i.e. X1 > X2) will plot values from right to left. Exchanging Y1 and Y2 (i.e. Y1 > Y2) will plot values with the minimum at the top with bars being drawn from the top also. Combining these actions with the Style (horizontal or vertical) gives all possible plot orientations.
The plot ignores invalid data elements in the array and leaves blank spaces for them on the display. If the number of points extends past the end of the array, the plot will continue at the first of the array until N points or all the points of the array have been plotted.

The plot area defined by X1, Y1, X2, and Y2 limits the area which can be plotted on the screen. Any points which fall outside this area will not appear on the screen but will be "clipped" at the boundary.

Example:

Given an array with 10 elements whose values range from 0 to 100.

```
IF Watch(1);
[
```

```
   Data = new(10);
   Data[0] = 0;
   Data[1] = 1;
   Data[2] = 20;
   Data[3] = 25;
   Data[4] = 26;
   Data[5] = 40;
   Data[6] = 55;
   Data[7] = 60;
   Data[8] = 75;
   Data[9] = 99;
]
```

A plot of the data may be done with the following:

```
Plot(data[0] { Starting element },
     10      { Number of elements to plot },
     0       { Low limit },
     100      { High limit },
     100, 150, 200, 50 { Bounding box for plot area, offset for 42
pixel title bar},
     1+10+50 { Style: Solid line style, vertical plot, step fashion
},
     4       { Color: dark red},
     0       { Pattern: Line plot (not bars) },
     0       { Pattern Background: Ignored for line plot });
```

This statement will result in a line plot of 10 data points in an area in the upper left corner of the window. The line will be vertically orientated meaning that Data[0] is at the bottom left corner and each successive array value is plotted one step higher, with values increasing to the right. It is a single line in style and dark red in color. It will follow a step pattern. That is, it will look like the outline of a bar graph of the points.

Example 2:

Plotting the same values as a horizontal plot, using Pen and Brush values:

```
Plot(Data[0],
     10,
     0,
     100,
     200, 150, 300, 50,
     Pen("<FFFFFF00>" { yellow }, 3 { dotted }, 2 { pixel width }),
     Brush("<FFFFFF>", 0, 2 { Stripe pattern });
     {, XORMode, Direction, DrawStepped, BitNumber, Average});
```

PlotBuff

Description: Displays a plot of a subsection of a buffer in a particular area of the window after converting the buffer to element values. Extends Plot.

Returns: Nothing

Usage: Steady State only.

Function Groups: Graphics, String and Buffer

Related to: Plot | PlotXY

Format: PlotBuff(Buffer, Offset, N, Type, Low, High, X1, Y1, X2, Y2, Style, Color, Pattern, Background)
...OR...
PlotBuff(Buffer, Offset, N, Type, Low, High, X1, Y1, X2, Y2, Pen, Brush [, XORMode, Direction, DrawStepped, BitNumber, Average])

Parameters:

Buffer

Required. Any text expression that contains data to be plotted.

Offset

Required. Any numeric expression that gives the starting buffer position in data elements (not bytes or characters) of the first data element.

N

Required. Any numeric expression giving the number of elements to plot starting at the element given by the offset parameter. N must be greater than or equal to 2, and no greater than 32000.

If this parameter is greater than the length of the buffer, the number of points plotted will be the number of points available in the buffer. If the end of the buffer is encountered before N points have been plotted, plotting continues with the first point in the buffer (offset 0).

Type

Required. Any numeric expression giving the type of data held in the buffer. The types are described in the following table:

Value	Type
0	Byte (0-255)
1	Short (2 bytes. -32768 to 32767)
2	Long (4 byte signed integer)
3	Float (4 byte IEEE floating point)

Low

Required. Any numeric expression that defines the limit of data values to be plotted. This value need not be less than High but must not be equal to it. In the standard case, when this value is less than High, it will define the minimum value to be displayed.
For example, if Low was 10, a data value of 10 would fall on the edge of the bounding box, on either X1, if the plot is vertical, or on Y1, if the plot is horizontal. Any values below 10 in this case would not be shown - they would be outside the clipped box.

High

Required. Any numeric expression that defines the limit of data values to be plotted. This value need not be greater than Low but must not be equal to it. In the standard case, when this value is greater than Low, it will define the maximum value to be displayed.
For example, if High was 1000, a data value of 1000 would fall on the edge of the bounding box, on either X2, if the plot is vertical, or on Y2, if the plot is horizontal. Any values above 1000 in this case would not be shown - they would be outside the clipped box.

X1

Required. Any numeric expression giving the X coordinate on the screen of one side of the plot area. This is typically, although not necessarily, the left side of the plot.

For horizontal plots, this is the screen coordinate of the first point or bar plotted. For vertical plots, this is the value which corresponds to the Low parameter value. For vertical bar plots, X1 is the base coordinate for the bars.

Y1

Required. Any numeric expression giving the Y coordinate on the screen of one side of the plot area. This is typically, although not necessarily, the bottom of the plot.

For vertical plots, this is the screen coordinate of the first point or bar plotted. For horizontal plots, this is the value which corresponds to the Low parameter value. For horizontal bar plots, Y1 is the base coordinate for the bars.

X2

Required. Any numeric expression giving the X coordinate on the screen of the side of the plot area opposite to X1. This is typically, although not necessarily, the right side of the plot.

For horizontal plots, this is the screen coordinate of the last point or bar plotted. For vertical plots, this is the value which corresponds to the High parameter value.

Y2

Required. Any numeric expression giving the Y coordinate on the screen of the side of the plot area opposite to Y1. This is typically, although not necessarily, the top of the plot. For vertical plots, this is the screen coordinate of the last point or bar plotted. For

horizontal plots, this is the value which corresponds to the High parameter value.

Style

Required. Any numeric expression giving the style of the plot – the line style, direction, bit number for bits plots, and the number of points to average per point displayed.

If PlotBuff senses that the Style parameter is a Pen, the Destructive, XORMode, DrawStepped, BitNumber, and Average parameters will become relevant. The Style is determined by adding together the following values.

Line style + Direction + Path + Bit number + Average

Line style is a number between 1 and 5 inclusive, giving the line style for line plots. A line style of 1 indicates a solid line. For bar plots, the line style portion of the Style parameter is ignored.
Direction is either 0 or 10, where 0 is a horizontal plot and 10 is a vertical plot.
Path is either 0 or 50. A 0 indicates that line plots are to be drawn directly from point to point. A 50 indicates that plots are to be drawn in a step fashion with two line segments between each point. The first segment runs parallel to the X-axis, the second, parallel to the Y-axis. This produces a square-looking plot. These step plots are useful for plotting status values which change in jumps rather than continuously. The Path value is ignored for bar plots.
Bit number is the number of the bit to use in the data for plotting. If the normal entire value of the data is to be plotted, use 0 for this value. If only one of the bits from the data is to be plotted, use the bit number plus

1. The value to add is (Bit number + 1) * 100 or 0. The ability to plot only a single bit from short or long values allows status data to be stored very efficiently by not requiring a separate buffer for each status value. For bit plots, the value plotted is always a 0 or a 1 corresponding to the value of the selected bit. This plotting option is usually done in conjunction with a Path value of 50 for step plots.

Average is the number of consecutive points to average to give a single point on the screen. It can be used to plot a very large number of values on the screen without having to draw all of the points on the screen. This will produce a smoothed plot and reduce the drawing time. The number to add is (Data points per displayed point - 1) * 10000.

Color

Required. Any numeric expression giving the VTScada Color Palette of the line and the foreground color for filled plots. If the number is less than 10000, the plot is non-destructive. If the number is greater than or equal to 10000, the plot is destructive and the actual color used is Color - 10000.

Pattern

Required. Any numeric expression giving the bar Fill Patterns for the plot. For a Pattern value of 0, the plot is a line plot. For Pattern values in the range of 1 to 25 inclusive, the plot is a bar plot with the hatch pattern corresponding to the Pattern parameter value. If the parameter is 1, the bars are a solid color and the Background parameter is ignored.

Background

Required. Any numeric expression giving the color number of the plot fill background color. This number is ignored if the Pattern parameter is equal to 1. For val-

ues of Pattern greater than 2, it gives the background color for the bar.

Pen

Required. The pen to be used (if any).

Brush

The brush to be used (if any).

XORMode

An optional parameter specifying whether or not the plot is destructive (if destructive, the plot line won't be affected by the background color). If set to 0 or Invalid (default), the destructive drawing mode will be used. If set to 1, the non-destructive or XOR drawing mode will be used.

Direction

An optional parameter specifying the horizontal or vertical direction for the plot. 0 or Invalid indicates horizontal, while 1 indicates vertical.

DrawStepped

An optional parameter specifying whether the plot is to be drawn from point-to-point, or stepped. 0 or Invalid indicates point-to-point, while 1 indicates stepped.

BitNumber

An optional parameter specifying the use of a whole value or a bit number. Invalid indicates a whole value should be used, while >=0 indicates that the specified bit number should be used.

Average

An optional parameter specifying whether or not consecutive points should be averaged to give a single point. Invalid or 0 indicates that averaging should not be used, while >0 indicates that the number of consecutive points should be averaged to give a single

point.

Comments:

This function is a layered graphics statement.
This statement will execute timer functions during the plotting process, allowing time critical functions such as driver I/O and data logging to continue running during the relatively long times it takes to update a plot on the screen.

Through the ordering of Low, High, X1, X2, Y1, and Y2, there are 8 different orientations possible for line plots and 16 for bar plots. The normal left to right plot with minimum at the bottom will have Low < High, X1 < X2, and Y1 < Y2. By exchanging Low and High (i.e. Low > High), the plot will have minimum at the top with bars drawn from the bottom. Exchanging X1 and X2 (i.e. X1 > X2) will plot values from right to left. Exchanging Y1 and Y2 (i.e. Y1 > Y2) will plot values with minimum at the top and bars drawn from the top also. Combining these actions with the Style (horizontal or vertical) gives all possible plot orientations.

If the number of points extends past the end of the buffer, the plot will continue at the first of the buffer until N points or all the points of the buffer have been plotted.
The plot area defined by X1, Y1, X2, and Y2 limits the area which can be plotted on the screen. Any points which fall outside this area will not appear on the screen but will be "clipped" at the boundary.

The PlotBuff statement works the same as the Plot statement. However, invalid values cannot exist in the buffer. The RAM memory savings of using buffers rather than arrays is in the range of 45% to 73%.

Example:

```
{ Create the buffer and store the data in it }
If ! Valid(data);
```

```
[
  data = MakeBuff(5, 0);
  SetByte(data, 0, 1);
  SetByte(data, 1, 9);
  SetByte(data, 2, 0);
  SetByte(data, 3, 6);
  SetByte(data, 4, 10);
]
PlotBuff(data { Buffer name },
         0, 10 { Start at first point and plot 10 points },
         0 { Data in the range of 0-255; a single byte },
         0, 10 { Low and high limits },
         100, 500, 600, 100 { Bounding box for clipped area },
         0 { Horizontal plot },
         12 { Light red },
         6 { Bar plot with vertical line pattern },
         4 { Dark red background for pattern });
```

This plots a bar graph of 5 points in an area centered in the window. The bars will be vertical (points run along the horizontal), with light red vertical line pattern on dark red background.

PlotXY

Description: Displays a plot of a curve in the window given the X and Y values in two arrays.

Returns: Nothing

Usage: Steady State only.

Function Groups: Graphics

Related to: Plot | PlotBuff | Sort

Format: PlotXY(XArrayElem, YArrayElem, N, LowX, HighX, LowY, HighY, X1, Y1, X2, Y2, Style, Color)
< OR >
PlotXY(XArrayElem, YArrayElem, N, LowX, HighX, LowY, HighY, X1, Y1, X2, Y2, Pen, Reserved [, XORMode, Direction, DrawStepped, BitNumber, Average])

Note: PlotXY takes the first parameter set or else senses that the Style parameter is a Pen, in which cast the second parameter set is used:

Parameters:

XArrayElem

Required. Any array element giving the starting point in the array of X coordinates. The subscript for the array may be any numeric expression. If processing a multidimensional array, the usual rules apply to decide which dimension should be used.

YArrayElem

Required. Any array element giving the starting point in the array of Y coordinates. The subscript for the array may be any numeric expression. If processing a multidimensional array, the usual rules apply to decide which dimension should be used.

N

Required. Any numeric expression giving the number of array elements to plot starting at the element given by the first parameters. N must be greater than or equal to 0, and no greater than 16384.

If this parameter is greater than the dimension of the first array, the number of points plotted will be that array dimension.

If N extends past the upper bound of the lowest array dimension, this computation will "wrap-around" and resume at element 0, until N elements have been processed.

LowX

Required. Any numeric expression that defines the limit of data values to be plotted. This value need not be less than HighX but must not be equal to it. In the standard case, when this value is less than HighX, it will define the minimum value to be displayed from the array of X-values.

For example, if LowX was 10, a data value of 10 would fall on the edge of the bounding box, on either X1, if the plot is vertical, or on Y1, if the plot is horizontal.

Any values below 10 in this case would not be shown – they would be outside the clipped box.

HighX

Required. Any numeric expression that defines the limit of data values to be plotted. This value need not be greater than LowX but must not be equal to it. In the standard case, when this value is greater than LowX, it will define the maximum value to be displayed from the array of X-values.

For example, if HighX was 1000, a data value of 1000 would fall on the edge of the bounding box, on either X2, if the plot is vertical, or on Y2, if the plot is horizontal. Any values above 1000 in this case would not be shown – they would be outside the clipped box.

LowY

Required. Any numeric expression that defines the limit of data values to be plotted. This value need not be less than HighY but must not be equal to it. In the standard case, when this value is less than HighY, it will define the minimum value to be displayed from the array of Y-values.

For example, if LowY was 10, a data value of 10 would fall on the edge of the bounding box, on either Y1, if the plot is vertical, or on X1, if the plot is horizontal. Any values below 10 in this case would not be shown – they would be outside the clipped box.

HighY

Required. Any numeric expression that defines the limit of data values to be plotted. This value need not be greater than LowY but must not be equal to it. In the standard case, when this value is greater than LowY, it will define the maximum value to be displayed from the array of Y-values.

For example, if HighY was 1000, a data value of 1000

would fall on the edge of the bounding box, on either Y2, if the plot is vertical, or on X2, if the plot is horizontal. Any values above 1000 in this case would not be shown - they would be outside the clipped box.

X1

Required. Any numeric expression giving the X coordinate on the screen of one side of the plot area. This is typically, although not necessarily, the left side of the plot. This corresponds to the LowX parameter for horizontal plots and to the LowY parameter for vertical plots.

Y1

Required. Any numeric expression giving the Y coordinate on the screen of one side of the plot area. This is typically, although not necessarily, the top of the plot. This corresponds to the LowY parameter for horizontal plots and to the LowX parameter for vertical plots.

X2

Required. Any numeric expression giving the X coordinate on the screen of the side of the plot area opposite to X1. This is typically, although not necessarily, the right side of the plot. This corresponds to the HighX parameter for horizontal plots and to the HighY parameter for vertical plots.

Y2

Required. Any numeric expression giving the Y coordinate on the screen of the side of the plot area opposite to Y1. This is typically, although not necessarily, the bottom of the plot. This corresponds to the HighY parameter for horizontal plots and to the HighX parameter for vertical plots.

Style

Required. Any numeric expression giving the style of the plot – the line style, direction, bit number for bits plots, and the number of points to average per point displayed. If PlotXY senses that the Style parameter is a Pen, XORMode, DrawStepped, BitNumber, and Average parameters will become relevant. The Style is determined by adding together the following values.

Line style + Direction + Path + Bit number + Average

Line style is a number between 1 and 10 inclusive which gives the line style for line plots. A line style of 1 indicates a solid line. For other line styles, the background color is set to the current background screen color.
Direction is either 0 or 10. A 0 indicates a horizontal plot and a 10 indicates a vertical plot.
Path is either 0 or 50. A 0 indicates that line plots are to be drawn directly from point to point. A 50 indicates that plots are to be drawn in a step fashion with two line segments between each point. The first segment runs parallel to the X-axis and the second segment runs parallel to the Y-axis. This produces a square-looking plot. These step plots are useful for plotting status values which change in jumps rather than continuously.
Bit number is the number of the bit to use in the YArray data for plotting. If the normal entire value of the data is to be plotted, use 0 for this value. If only one of the bits from the data is to be plotted, use the bit number plus 1. The value to add is (Bit number + 1) * 100 or 0. The ability to plot only a single bit from an array of short or long values allows status data to be stored very efficiently by not requiring a separate array for each status value. For bit plots, the value plotted is

always a 0 or a 1 corresponding to the value of the selected bit. The use of this plotting option is usually done in conjunction with a Path value of 50 for step plots.

Average is the number of consecutive array points to average to give a single point on the screen. Both XArray and YArray values are averaged. It can be used to plot a very large array of values on the screen without having to draw all of the points on the screen. This will produce a smoothed plot and reduce the drawing time. The number to add is (Data points per displayed point + 1) * 10000.

Color

Any numeric expression giving the VTScada Color Palette of the line and the foreground color for filled plots. If the number is less than 10000, the plot is non-destructive.

If the number is greater than or equal to 10000, the plot is destructive and the actual color used is Color - 10000.

Pen

The pen to be used (if any).

Reserved

Reserved has been allocated to permit the addition of a Brush parameter, should such a parameter become necessary in the future. Reserved can currently be any value or Invalid.

XORMode

An optional parameter specifying whether or not the plot is destructive (if destructive, the plot line won't be affected by the background color).

If set to 0 or Invalid (default), the destructive drawing mode will be used. If set to 1, the non-destructive or XOR drawing mode will be used.

Direction

An optional parameter specifying the horizontal or vertical direction for the plot. 0 or Invalid indicates horizontal, while 1 indicates vertical.

DrawStepped

An optional parameter specifying whether the plot is to be drawn from point-to-point, or stepped. 0 or Invalid indicates point-to-point, while 1 indicates stepped.

BitNumber

An optional parameter specifying the use of a whole value or a bit number. Invalid indicates a whole value should be used, while >=0 indicates that the specified bit number should be used.

Average

An optional parameter specifying whether or not consecutive points should be averaged to give a single point.
Invalid or 0 indicates that averaging should not be used, while >0 indicates that the number of consecutive points should be averaged to give a single point.

Comments:

This function is a layered graphics statement.
The PlotXY statement will execute timer functions during the plotting process. This will allow time critical functions such as driver I/O and data logging to continue running during the relatively long times it takes to update a plot on the screen.

Through the ordering of the LowX, HighX, LowY, HighY, X1, X2, Y1, and Y2 parameters, the plot may have 8 different orientations. The normal left to right orientation with the minimum at the bottom will have LowX < HighY, LowY < HighY, X1 < X2, and Y1 > Y2.

Exchanging either LowX and HighX (i.e. LowX > HighX) or X1 and X2 (i.e. X1 > X2), the plot will be from right to left. This means that the minimum X-value will be on the right rather than the left. Exchanging either LowY and HighY (i.e. LowY > HighY) or Y1 and Y2 (i.e. Y1 > Y2),will result in the plot will having its minimum Y-value at the top rather than the bottom.
PlotXY ignores invalid data elements in the arrays and leaves blank spaces for them on the display. If the X-values are not in order, the plot will appear as a scatter of lines. The Sort statement may be used to order the points in the arrays prior to plotting.

The plot area defined by X1, Y1, X2, and Y2 limits the area which can be plotted on the screen. Any points which fall outside this area will not appear on the screen but will be "clipped" at the boundary.

Example:

Two arrays called speed and efficiency exist that each have 100 elements. The array called speed has values that range from 0 to 1800, while efficiency has values ranging from 0 to 100 (percent). A plot of speed against efficiency may be done with the following:

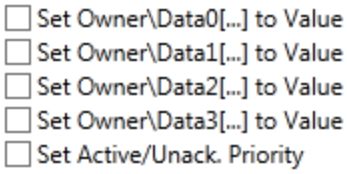
```
PlotXY(speed[0] { Starting element of X-values },
       efficiency[0] { Starting element of Y-values },
       100 { Number of elements to plot },
       0, 1800 { Low/high limits for speed },
       0, 100 { Low/high limits for efficiency },
       0, 500, 500, 0 { Bounding box for clipped area },
       1 { Solid line style, horizontal plot },
       15 { Plot is white });
```

This statement will result in a line plot of 100 data points in an area in the upper left corner of the window. The line will be horizontally orientated, solid in style and white in color.

PMultiCheckBox

(Dialog Library)

Description: Tool used to display the standard set of check boxes. Commonly used in the Owner tab of a configuration folder.



Returns: Nothing

Usage: Steady State only.

Function Groups: Graphics

Related to: GUITransform | PAddressEntry | PAreaSelect | PCheckBox | PContributor | PDroplist | PEditfield | PPageSelect | PRadioButtons | PSecBit | PSelectObject | PSpinbox | PTypeToggle

Format: \DialogLibrary\PMulticheck box(ParmNum [, LeftToRight, CheckBoxLabels, FocusID, DrawBevel, Label, BitOffset])

Parameters:

ParmNum

Required. Any numeric expression giving the parameter number (from 0) in the caller to alter.

LeftToRight

An optional parameter that is any logical expression. If TRUE, the ordering will be left-to-right instead of top-to-bottom. Defaults to FALSE.

CheckBoxLabels

An array of labels for the check boxes.

FocusID

Boolean. If this value is FALSE (0), the field will display

its current setting, but cannot be opened (i.e. its value cannot be changed), and will appear disabled (grayed-out).

DrawBevel

An optional flag indicating whether a bevel should be drawn around the set of check boxes. Defaults to FALSE.

Label

An optional title to be displayed above the check boxes.

BitOffset

An offset at which to start bit-setting.

Comments:

This tool handles the shortcut menu and optionally highlights it's entire area when the an override is specified. Feedback is also provided to allow custom controls to be highlighted in a more attacrive fashion.

This module is a member of the VTScada Dialog Library and must therefore be called from within a GUITransform and prefaced by \DialogLibrary\.

The "P" tools (Pcheck box, PContributor, PColorSelect, PDroplist, and PEditfield) were intended only for use in configuration folders and drawing panel modules, and therefore are subject to the system security restraints.

This parameter tool expects the first parameter of its calling module to contain an array of tag parameters.

Usual height: 155 pixels.

Example:

```
HasPv = \SecurityManager\SecurityCheck(\Se-
curityManager\PrivBitTagModify, 1);
GUITransform(40, 203, 460, 115,
             1, 1, 1, 1, 1           { No scaling
},
             0, 0, 1, 0              { No movement; visible;
reserved },
             0, 0, 0                 { Not selectable
},
             \DialogLibrary\PMulticheck box(\#ContributionType,
                                            0,
                                            SetOwnerLabels,
                                            HasPv ? 3 : 0   { ID },
                                            0               {
DrawBevel },
                                            Invalid         { Label
}));
```

Point

Description: Returns a two-dimensional point, or location, in a window.

Returns: Point

Usage: Steady State only.

Function Groups: Graphics

Related to: Path | Rotate | Trajectory | Vertex

Format: Point(X, Y, Rotation, Trajectory)

Parameters:

X

Required. A numeric constant, that describes the reference x-axis location of the point. Expressions are not permitted here. If it is desired to change the x coordinate, use the Trajectory or Rotation parameters.

Y

Required. A numeric constant, that describes the reference y-axis location of the point. Expressions are not permitted here. If it is desired to change the y coordinate, use the Trajectory or Rotation parameters.

Rotation

Required. Any expression that returns a Rotate value. This specifies any translation of this point from its reference position (X, Y) by rotation about another point. If this is invalid, no rotation is performed, but the Point is still valid.

Trajectory

Required. Any expression that returns a Trajectory value. This specifies any translation of this point from its reference position of (X, Y) along a path. If this is invalid, no translation is performed, but the Point is still valid.

Comments: Points are used in the Rotate and Vertex functions.

Examples:

```
centerPt = Point(489, 122.5, INVALID, INVALID);
startAnglePt = Point(609, 27, INVALID, INVALID);
endAnglePt = Point(586, 243, INVALID, INVALID);
```

This defines 3 points that are neither translated nor rotated. They can be used in a Vertex or a Rotate function, like in the following example.

```
GUIPie(395, 187, 583, 58 { Bounding box for pie },
       1, 1, 1, 1, 1, 0, 0 { No scaling, trajectory or rotation },
       1, 0 { Pie is visible; reserved },
       0, 0, 0 { Cannot be focused/selected },
       12, 15 { Red fill outlined in white },
       Vertex(1 { Double smooth },
       centerPt, startAnglePt, endAnglePt));
```

PointerToBuff

Description: Returns a buffer containing the numeric data from the variables pointed at by each element of the array.

Returns: Buffer

Usage: Script or steady state.

Function Groups: Array, String and Buffer

Related to: BuffToArray | BuffToParm | BuffToPointer | GetByte | ParmToBuff | SetByte

Format: PointerToBuff(ArrayStart, N, Option, Size, Skip)

Parameters:

ArrayStart

Required. Any expression that specifies the starting array element. This array contains the pointers to the variables whose values will be written to a buffer. If processing a multidimensional array, the usual rules apply to decide which dimension should be used.

N

Required. Any numeric expression giving the number of array elements to convert starting at the element given by the first parameter.
If N extends past the upper bound of the lowest array dimension, this computation will "wrap-around" and resume at element 0, until N elements have been processed.

Option

Required. Any numeric expression that specifies the format of the buffer write:

Option	Format
0	Unsigned binary (low byte first)
1	Signed binary (low byte first)
2	BCD (binary coded decimal) (low byte first)
3	ASCII octal (high byte first)
4	ASCII decimal (high byte first)
5	ASCII hex (high byte first)
6	ASCII floating point (high byte first)
7	IEEE float/double (low byte first)
8	<obsolete>
9	Allen-Bradley® PLC/3 floating point
10	VAX single precision floating point

For Options 7 and 9 the data is written as appropriate binary format.

Size

Any numeric expression giving the number of digits in

Option	Size Meaning	Size Range
Binary types	Number of bits	1 - 32 bits
BCD	Number of 4-bit digits	1 - 8 digits
ASCII types	Number of bytes	1 - 32 bytes
Float types	Precision	1 for single precision, 2 for double precision

Skip

Any numeric expression giving the number of buffer bits/digits/bytes to skip after writing each non-floating point element. For floating point types, this parameter must be set to 0.

Comments: This function may only be used with pointers pointing at numeric data. Any invalid array elements, or invalid data pointed at by array elements are written to the buffer as formatted zeroes. This function is useful for writing I/O drivers and saving arrays of data in RAM with a fraction of the memory requirement.

Example:

```
buff1 = PointerToBuff(dataPtr[0]{ Starting point in array },
                      100 { No. of elements to convert },
                      5 { Write as ASCII hex },
                      4 { Number of bytes per },
                      0 { No skip between writes });
```

This produces a formatted buffer 400 bytes long (100 four byte groups). Each 4 bytes is a hex number corresponding to a variable pointed at by an element in the array. Invalid array elements and invalid data are written as 0.

PointList

Description: Returns an array of tag names within the current scope, given the name of a tag type or group.
NOTE: GetTagList should be used in place of PointList for all new code.

Returns: Array

Usage: Script Only.

Function Groups: Basic Module

Related to: GetTagList | ListVars

Format: \PointList(TagType[, SearchString, NoFilterSimulation])

Parameters:

TagType

Required. Any text expression giving a type of tag or the name of a tag group.

SearchString

An optional text expression. The search will be limited to tags with matching names. May include wildcards. The default is "*" (any name).

NoFilterSimulation

Obsolete.

Comments: This utility function reduces the effort required to obtain a list of tags of a particular type or group membership. If no matching tags are found, the first element of the array will be Invalid.

Examples:

```
{ Get a list of the Stations }
 Stations =  \PointList("LiftstationDrivers", "*", 0);
 NStations = ArraySize(Stations, 0);
```

Popup

Note: Deprecated. Do not use in new code.

(Alarm Manager module)

Description: Causes an alarm pop-up dialog to be displayed.

Returns: Numeric

Usage: Script Only.

Function Groups: Alarm, Graphics

Related to:

Format: \AlarmManager\Popup(ModuleName, Title, Width, Height, X, Y);

Parameters:

ModuleName

Required. Any expression providing a text result. Name of module to draw in the window

Title

Required. Any expression providing a text result. Window title

Width

Required. Numeric expression. Number of pixels wide for the dialog

Height

Required. Numeric expression. Number of pixels high for the dialog

X

Required. Numeric expression. Location on the screen for the dialog, measured from the left.

Y

Required. Numeric expression. Location on the screen for the dialog measured from the top.

Comments: The Popup subroutine always returns "1".

POverride

(Dialog Library)

Description: Tool used to override non-standard ConfigFolder controls for child tags.

Returns: Nothing

Usage: Steady State only.

Function Groups: Graphics

Related to: GUITransform | PAddressEntry | PAreaSelect | PCheckBox | PContributor | PDroplist | PEditfield | PPageSelect | PRadioButtons | PSecBit | PSelectObject | PSpinbox | PTypeToggle

Format: \DialogLibrary\POverride(ParmNum [, ShowHotbox, ShowHighlight, HighlightColor, IntTrigger, ShowProperties, ValType, MinVal, MaxVal, ValArray])

Parameters:

ParmNum

Required. Any numeric expression giving the parameter number (from 0) in the caller to alter.

ShowHotbox

An optional parameter that is any logical expression. If TRUE, the click-able area will be shown.

ShowHighlight

An optional parameter that is any logical expression. If TRUE the area will be emphasized with the HighlightColor.

HighlightColor

An optional output giving the highlight color to be displayed if ShowHighlight is true.

IntTrigger

An optional trigger used by the caller's Edit field.

ShowProperties

An optional Boolean, controlling whether the Properties item should appear in the menu.

ValType

Optional. The VTScada value type of the parameter.

MinVal

Optional numeric. The minimum for the parameter value.

MaxVal

Optional numeric. The maximum for the parameter value.

Comments: This tool handles the shortcut menu and optionally highlights it's entire area when the an override is specified. Feedback is also provided to allow custom controls to be highlighted in a more attractive fashion.
This module is a member of the VTScada Dialog Library and must therefore be called from within a GUITransform and prefaced by \DialogLibrary\.
The "P" tools (PCheckBox, PContributor, PColorSelect, PDroplist, and PEditfield) were intended only for use in configuration folders and drawing panel modules, and therefore are subject to the system security restraints.
This parameter tool expects the first parameter of its calling module to contain an array of tag parameters.
Usual height: 45 pixels.

Example:

```
GUITransform(30, 145, WIDTH/2 - 5, 100,
             1, 1, 1, 1, 1,
             0, 0, 1, 0,
             0, 0, 0,
             \DialogLibrary\POverride(\#Address,
                                      0,
                                      0,
                                      HighlightColor,
                                      PickValid(Trigger1, 1) &&
                                      PickValid(Trigger2, 1) &&
                                      PickValid(Trigger3, 1)));
```

Pow

Description: Returns a number raised to a power.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Generic Math

Related to: Exp | Ln | Log

Format: Pow(X, Exponent)

Parameters:

X

Required. Any numeric expression giving the number to be raised to the power of Exponent.

Exponent

Required. Any numeric expression giving the number to which X should be raised.

Comments: This function may be used to perform the common anti-logarithm by using 10 for X. The return value is invalid if X is less than or equal to 0 and Y is not an integer.

Examples:

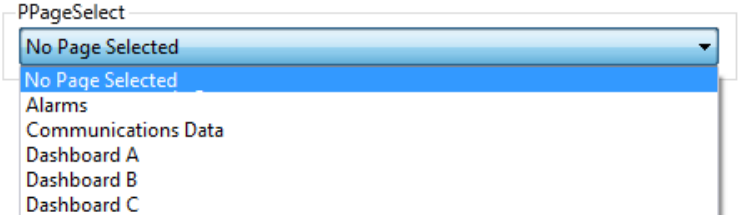
```
r = Pow( 2, 4);
s = Pow(10, 2);
t = Pow(64, 0.5);
```

The values of r, s, and t will be 16, 100, and 8 respectively.

PPageSelect

(Dialog Library)

Description: Draws a titled, beveled droplist of pages in the system.



Returns: Nothing

Usage: Steady State only.

Function Groups: Graphics

Related to: GUITransform | PAddressEntry | PAreaSelect | PCheckBox | PColorSelect | PContributor | PDroplist | PEditfield | PRadioButtons | PSecBit | PSelectObject | PSpinbox | PTypeToggle

Format: \DialogLibrary\PPageSelect(ParmNum [, Title, FocusID, VertAlign, AlignTitle])

Parameters:

ParmNum

Required. Any numeric expression giving the parameter number (from 0) in the caller to alter.

Title

An optional parameter that is any text expression to be used as a title for the droplist.

FocusID

Boolean. If this value is FALSE (0), the field will display its current setting, but cannot be opened (i.e. its value cannot be changed), and will appear disabled (grayed-out).

VertAlign

An optional parameter that is any numeric expression that sets the vertical alignment of the droplist according to one of the following options:

VertAlign	Vertical Alignment
0	Top
1	Center
2	Bottom

Whether or not the title is included when the vertical alignment is calculated is determined by the value of AlignTitle. The default value is 0.

AlignTitle

An optional parameter that is any logical expression. If true (non-0) the title is included in the calculation for vertical alignment, if false(0) it is added to the droplist after it and its bevel has been vertically aligned. The default is true.

Comments:

This module is a member of the VTScada Dialog Library and must therefore be called from within a GUITransform and prefaced by \DialogLibrary\.
This parameter tool expects the first parameter of its calling module to contain an array of tag parameters. It will then set the element indicated by ParmNum to the name of the chosen page.
The height of the (unopened) droplist is constant, with the horizontal boundaries of its calling transform defining its width, and the vertical boundaries of its calling transform defining its opened height, which will include the added height of the bevel above the field, but may or may not include the title, depending on the alignment used. Note that if the entire list can be displayed in a smaller area than indicated by the vertical boundaries of the calling transform, the dropped list height will be decreased. The dropped height of the list will always have a minimum height of 1 line (below the field).
For any optional parameter that is to be set, all optional parameters preceding the desired one must be present, although they may be invalid.
Usual height: 45 pixels.

Examples:

```
GUITransform(10, 125, 210, 25,
             1, 1, 1, 1, 1,
             0, 0, 1, 0,
             0, 0, 0,
             \DialogLibrary\PPageSelect(2 { Parm num },
             "Select Page" { Title },
             3 { Focus ID },
             Invalid { Default alignment },
             1 { Align title }));
```

Notice in the above example that the last two parameters could have been omitted, because they are optional and their settings match those used as a default.

PPPDial

Description: Creates a PPP connection to a remote peer. The connection can be via a dial-up or direct device connection.

Returns: PPPhandle if succesful, otherwise an error value.

Usage: Script Only.

Function Groups: Modem, Serial Port

Related to: PPPHandles | PPPStatus

Format: PPPDial(DeviceName[, PhoneNumber, UserName, Password, Domain, Flags, DeviceType, DeviceSubType])

Parameters:

DeviceName

Required. Any text expression giving the name of the modem as configured in Windows™. The function will return Invalid if this parameter is not a valid text value.

PhoneNumber

An optional text expression giving the phone number to be used when making a PPP call. Required only for use with modems. Defaults to Invalid.

UserName

An optional text expression, providing the user name for remote authentication.

Password

An optional text expression, providing the password for remote authentication.

Domain

An optional text expression, providing the domain name for remote authentication.

Flags

An optional numeric expression, specifying compression, encryption and other options to be provided. A bit-wise AND operation is done between this value and the following constants.

Value	Constant	Meaning
0x0001	PPPDF_IpHeaderCompression	Use IP header compression
0x0002	PPPDF_SwCompression	Use software compression
0x0004	PPPDF_DisableLcpExtensions	Disable LCP extensions.
0x0008	PPPDF_RequireEncryptedPw	Require encrypted password.
0x0010	PPPDF_RequireDataEncryption	Require data encryption

DeviceType

An optional numeric expression identifying the class of RAS device referred to by DeviceName. Possible values are
1 (Phone),
2 (VPN),
3 (Direct - serial/parallel/USB),
4 (Internet) and

5 (Broadband).
To create a PPPoE connection, use 5.

DeviceSubType

An optional text expression specifying the type of device referred to by DeviceName. Possible values are "modem", "isdn", "x25", "vpn", "pad", "GENERIC", "SERIAL", "FRAMERELAY", "ATM", "SONET", "SW56", "IRDA", "PARALLEL" and "PPPoE".

Comments: PPPDial works by creating a phonebook entry in a temporary phone book in the installation folder. The entry is GUID named and has a lifespan the same as the PPPHandle this method creates. A RAS call is then initiated and a PPPHandle created to represent the call.
This function looks for the subroutine, PPPStatus in the scope of the caller. After initiation of the call, this subroutine is called to notify the caller of the progress and status of the call.
The call is terminated and the phonebook entry removed once the PPPHandle destructs. Invalidating the last reference will do this. Destruction is asynchronous. VTScada shutdown waits for all PPP handles to gracefully close.

PPPHandles

Description: Returns an array of all Point-to-Point Protocol handles on the local machine. This includes all such handles, whether inbound or outbound and whether or not made by VTScada.

Returns: An array of PPP handles

Usage: Steady State only.

Function Groups: Modem, Serial Port

Related to: PPPDial | PPPStatus

Format:	PPPHandles()
Para-meters:	None

Comments: A PPP handle is a structure, defined as follows.

To access these values, use the PPPStatus command as follows:

```
PPPStruct = PPPStatus(Handles[I]);
rr1 = PPPStruct\ConnectionID;
```

Variable	Contents
ConnectionID	Only valid for outbound connections. The GUID used in the temporary phone book used to dial the call. PPPDial creates a temporary RAS phone book in the Data\Temp folder of the VTScada installation folder. It is removed on graceful exit of VTScada. Each outbound call is given a unique call ID (a GUID) that is entered into the phone book.
State	A numeric value identifying the state of the connection. There are many intermediate states, but the significant ones are 2 (dialling), 8192 (connected) and 8193 (disconnected).
Error	Zero indicates no errors. When non-zero, this holds a standard RAS error code, normally in the range 600-800.
RemoteIP	The IP of the remote end of a connected or disconnected connection.
LocalIP	The IP of the local end of a connected or disconnected connection. The IP is not invalidated once set, so that script code can handle transient connections.
DeviceName	The same as the DeviceName parameter of PPPDial. This is the Windows-assigned friendly name for the device that the connection is made through.

PPPStatus

Description: Obtains the structure of a PPP connection.

Returns: A structure describing the state. See comments

Usage: Script or steady state.

Function Groups: Modem, Serial Port

Related to: PPPDial | PPPStatus

Format: PPPStatus(PPPHandle)

Parameters:

PPPHandle

The PPP Handle to be monitored.

Comments: If called in script mode, the current state will be returned. If called in steady state, the function will be retriggered each time there is a state change in the parameter, PPPHandle.

The return value is a structure containing the following members:

ConnectionID: Actually the phonebook entry name, which is a GUID.

State: See enumeration RASCONNSTATE (defined in the Visual Studio file, ras.h)

Error: Last error code from RAS (defined in the Visual Studio file, raserror.h)

RemoteIP: The IP address of the remote peer. Only valid when State is RASCS_Projected or greater.

LocalIP: The IP address on this machine that the remote peer is connected to. Only valid when State is RASCS_Projected or greater.

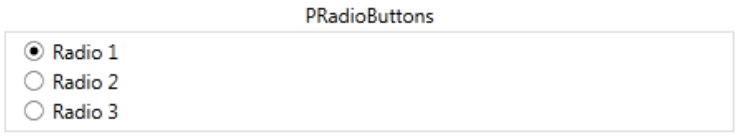
If the PPPHandle parameter is bad, this function will return an Invalid.

Related Functions:

PRadioButtons

(Dialog Library)

Description: Parameter Setting Radio Buttons. This module draws a set of labeled radio buttons with optional title and border.



Returns: Nothing

Usage: Steady State only.

Function Groups: Graphics

Related to: GUITransform | PAreaSelect | PCheckBox | PColorSelect | PContributor | PDroplist | PEditfield | PPageSelect | PSecBit | PSelectObject | PSpinbox | PTypeToggle

Format: \DialogLibrary\PRadioButtons(ParmNum, FocusID, Border, Title, Options, AlignTitle, L1 [, L2, ..., L16])

Parameters:

ParmNum

Required. Any numeric expression giving the parameter number (from 0) in the caller to alter.

FocusID

Boolean. If this value is FALSE (0), the field will display its current setting, but cannot be opened (i.e. its value cannot be changed), and will appear disabled (grayed-out).

Border

Any logical expression. If true (non-0) the buttons will have a raised border, if false (0) there will be no border around the buttons.

Title

Any text expression to be used as a title with the radio buttons. The default value is to have no title.

Options

A bit-wise expression.
Bit 0 controls left/right placement of the buttons. If set, buttons will be placed to the left of the labels.
Bit 1 controls vertical/horizontal orientation. If set, the radio buttons will be displayed horizontally. Bit 0 has no effect while bit 1 is set.
Defaults to 0 – neither bit set.

AlignTitle

Any logical expression; if true (non-0) the title is drawn within the radio buttons' boundaries, if false(0) the buttons fill their bounding area and the title is added at the top (i.e. it extends past the top boundary). The default is true.

L1, L2, ...L16

Labels. The number of labels determines the number of buttons displayed. There must be at least two labels.
L1 may be an array of text labels, or the first of a series of up to sixteen individual text labels for the radio buttons.
If L1 is an individual label, then up to L16 individual labels may follow.
If L1 is an array of labels, L2 may (optionally) be defined as an array of Return values, similar to the return array in PDropList. No further parameters may follow.

Comments: This module is a member of the VTScada Dialog Library and must therefore be called from within a GUITransform and prefaced by \DialogLibrary\.

This parameter tool expects the first parameter of its calling module to contain an array of tag parameters. It will then set the element indicated by ParmNum to the selected item.
The boundaries of the calling transform define the outline of the buttons including their border, if there is one. If the area is too small to fully display the buttons they will extend beyond their right and bottom boundaries. Buttons and their border will not overlap each other and will always be shown in their entirety, although the labels may be clipped or entirely deleted.
Usual height: varies according to number of buttons. Seldom less than 50px.

Example:

```
GUITransform(50, 400, 450, 100,
             1, 1, 1, 1, 1,
             0, 0, 1, 0,
             0, 0, 0,
             \DialogLibrary\PRadioButtons(7 { Parm num },
             2 { Focus ID },
             1 { Draw a border },
             "" { No title required },
             0 { Buttons on right },
             Invalid { Not req'd (no title) },
             "On", "Off", "Reverse", "Out of Service"
             { Labels }));
```

Related Functions:

Print

Description: Allows text to be printed.

Returns: Nothing

Usage: Script Only.

Function Groups: Printer

Related to: FWrite | PrintLine | PrtScrn | Redirect | SWrite

Format: Print(PrinterSpec, Text)

Parameters:

PrinterSpec

...will accept any of the following:

- Local Printer:
- Port name (including virtual ports) with or without a trailing colon (e.g. DEF or DEF:. COM1 or COM1:; USB001 or USB001:; etc.)
- Windows printer share (e.g. "XYZ Laser Printer")
- Windows share name (if the printer is shared) (e.g. "XYZLaser")
- Local or Remote Printer:
- UNC share name (which includes the host and share name (e.g. "\\localhost\XYlaser" or "\\lab1\NetPrinter")

Text

Required. Any text expression that gives the text to print. Control characters may be included in the text.

Comments: This statement is very similar to the PrintLine statement except it does not add the carriage return or line feed after the text.
All print functions are compatible with the values returned in either of the first two parameters of the PrintDialogBox function.

Example:

```
If 1 NextScreen;
[
  Print(1, "Daily Report: ");
]
```

This prints the string "Daily Report: " to LPT1:.

Related Functions:

PrintDialogBox

Description: Displays a threaded system common printer selection dialog box.

Returns: Nothing

Usage: Script Only.

Function Groups: Graphics, Printer

Related to: FontDialog | FileDialogBox | FileStream

Format: PrintDialogBox(PrinterName, PrinterDevice, MinPage, MaxPage, FirstPage, LastPage, NumCopies, Result [, AllowSelection])

Parameters:

PrinterName

Required. Any text expression giving the initial printer name to display as selected. If this is not a valid text value, the default printer (as configured under Windows) will initially be selected.
If PrinterName refers to a variable and the user clicks the "Print" button on the dialog, thereby closing it, the variable will be set to the name of the selected printer. If the user cancels the dialog box, the variable is unchanged.

PrinterDevice

Required. Any variable that will be set to the device name for the selected printer when the user clicks the "Print" button on the dialog.
If the user cancels the dialog box, the variable is unchanged. The text expression returned here can be supplied to a FileStream statement to open a stream to the selected printer.

MinPage

Required. Any positive integer expression specifying

the minimum page number in the printed output (see Comments section for further details).

MaxPage

Required. Any positive integer expression specifying the maximum page number in the printed output (see Comments section for further details).

FirstPage

Required. Any integer expression specifying the first page to be printed (see Comments section for further details).

LastPage

Required. Any integer expression specifying the last page to be printed (see Comments section for further details).

NumCopies

Required. Any integer expression specifying the number of copies to be printed (see Comments section for further details).

Result

Required. A variable that will be set to one of the following values:

- Invalid when this statement is run,
- Zero (0) to indicate that the user has cancelled the dialog, or
- One (1) to indicate that the user has clicked the "Print" button.

AllowSelection

An optional parameter. If provided, it must be a non-constant variable. This enables and controls the "Selection" radio button in the print dialog box. See the Comments section for further details.

Comments:

This function creates a printer selection dialog that runs in its own thread, thereby avoiding blocking the calling code. The calling code can use the Result parameter to determine when the user has completed the dialog box (see Result above).

In addition to the Result parameter, the function itself will return an error code to indicate whether the dialog was successfully opened. A "1" indicates failure to open while a "0" indicates success.

The caller of this function is expected to supply the MinPage and MaxPage parameters, representing the minimum and maximum page number from which the user may select. FirstPage and LastPage can be initially set to any values between MinPage and MaxPage. If the combination of the four values is not logical (e.g. LastPage is smaller than FirstPage), VTScada will force logical values for the dialog. When Result is set to one, FirstPage and LastPage will contain the page range that the user selected. If Result is set to zero (i.e. the user has cancelled the dialog), FirstPage and LastPage are left at their initial values. Setting MinPage and MaxPage to Invalid or to equal values will disable the page range field. NumCopies can likewise contain an initial value, and will be set to the user-selected number of copies if Result returns a value of one.

The initial value of the Selection radio button is controlled by the value of the variable provided for the AllowSelection parameter:

- If the variable's value evaluates to a non-zero positive number, the Selection radio button will be selected.
- If the variable's value is zero, the radio-button will initially be unselected.

When the Print button on the print dialog box is

clicked, this variable receives a zero if the Selection radio-button was not selected and one if it was selected.

> Note: Within an Anywhere Client session, this function does nothing.

Example:

```
If ZButton(10, 90, 110, 120, "Browse Print", 1, System\DefFont);
[
  PrintDialogBox(PrinterName, PrinterDevice, MinPage, MaxPage,
                 FirstPage, LastPage, NumCopies, Result);
]
ZText(120, 112,
      Concat("Printer chosen: ", PickValid(PrinterName, "Invalid"),
             " (", PickValid(PrinterDevice, "Invalid"), ") Page ",
             PickValid(FirstPage, "Invalid"), " to ",
             PickValid(LastPage, "Invalid"), " Copies: ",
             PickValid(NumCopies, "Invalid")),
      0, System\DefFont);
```

PrintLine

Description: Allows text to be printed and is followed by a carriage return-line feed to the printer.

Returns: Nothing

Usage: Script Only.

Function Groups: Printer

Related to: FWrite | Print | PrtScrn | Redirect | SWrite

Format: PrintLine(PrinterSpec, Text)

Parameters:

PrinterSpec

...will accept any of the following:

- Local Printer:
- Port name (including virtual ports) with or without a trailing colon (e.g. DEF or DEF:. COM1 or COM1:; USB001 or USB001:; etc.)

- Windows printer share (e.g. "XYZ Laser Printer")
- Windows share name (if the printer is shared) (e.g. "XYZLaser")
- Local or Remote Printer:
- UNC share name (which includes the host and share name (e.g. "\\localhost\XYlaser" or "\\lab1\NetPrinter")

Text: Required. Any text expression that gives the text to print. Control characters may be included in the text.

Comments: This statement adds a carriage return and line feed to the end of the text so that the printer goes to the start of a new line after the text is printed.
All print functions are compatible with the values returned in either of the first two parameters of the PrintDialogBox function.

Example:

```
If 1 NextScreen;
[
  PrintLine(1, "Daily Report: ");
]
```

This prints the string "Daily Report: " to LPT1:, followed by a carriage return and a line feed.

Priority

Description: Sets the execution priority for a module, variable or object.

Returns: Nothing

Usage: Script Only.

Function Groups: Advanced Module, Variable

Related to: Pending | PriorityWeight

Format: Priority(Data, PriorityNum)

Parameters:

Data

Required. The module, variable, or object value to set the priority for.

PriorityNum

Required. A numeric value that sets the priority as indicated:

PriorityNum	Priority
0	Normal
1	High
2	Default

Default priority is only valid for modules and variables. It causes variables to assume the priority of the instance in which they occur.

Comments:

This function sets the execution priority for a value so that any statements which use that value will execute before other statements triggered from normal priority values. If Priority is applied to a module, all instances of that module will have that priority unless otherwise set. Priority for an object value will set all values in an instance and all module instances called from that instance to that priority unless otherwise set. Setting priority for a variable will set the priority for all instances of that variable and will override the priority setting of the module or instance in which it is contained.

Priority can be used to improve the performance of I/O drivers, alarm modules, control modules and other modules for which increased responsiveness is required. Use Priority sparingly since its results are only effective if there are non-priority values which can be preempted to improve the performance.

This function returns the previous priority of the object.

Example:

```
prevPriority = Priority(Self(), 1);
```

This sets the priority of the module that contains this statement to a high priority.

PriorityWeight

Description: Sets the system-wide weighting for priority values.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Software and Hardware

Related to: Pending | Priority

Format: PriorityWeight(NumStatements)

Parameters:

NumStatements

Required. A numeric value that specifies the number of priority triggered statements that will execute before a normal priority statement executes. The default value at startup is 10.

Comments: This function returns the previous priority weight if the parameter is valid, and invalid otherwise.

Example:

```
prevPriorWt = PriorityWeight(5);
```

This ensures that only 5 high priority statements execute before a normal priority statement is allowed to execute.

ProcInfo

Description: Returns basic information about the VTScada process.

Returns: Numeric

Usage: Script Only.

Function Groups: Software and Hardware

Related to:

Format: ProcInfo(DataType)

Parameters:

DataType

Required. A numeric value that specifies the type of process information to be retrieved by the function.

DataType	Function returns
0	Process ID
1	GDI Handle Count
2	User Handle Count

Comments: Used by the workstation status tag.

Example:

```
GDIHndl = ProcInfo(1);
```

Profile

Description: Returns an array profiling the execution of statements in the application.

Returns: Array – see comments

Usage: Script Only.

Function Groups: Software and Hardware

Related to: ThreadList | Watch | WatchArray

Format: Profile(StartStop)

Parameters:

StartStop

Required. A numeric value of 0 or 1 that toggles the function on or off. A value of 1 causes the function to begin profiling the system, while a value of 0 stops the profiling and causes the return value to be set.

Comments: This function returns a two-dimensional array with each column representing a different statement and each row containing the following information:

Row	Data
0	Count of a statement
1	Statement value
2	Approximate execution time of statement in milliseconds

It is important to note that in the case of a script executed by a trigger statement, the sum of the execution times for the script statements may be less than the execution time recorded for the trigger statement. This is because the script statements may be of sufficient duration to lose their processor time-slice and be suspended for a short time. While the suspended statement will have its timings adjusted to account for this, the trigger statement will not have any adjustments made.

Example:

```
If MatchKeys(2, "g");
[
  Profile(1);
]
If MatchKeys(2, "s");
[
  whatHappened = Profile(0);
]
```

These two scripts start and stop the profiling when the user presses the appropriate keys.

ProgressBar

(System Library)

Description: Displays a horizontal progress bar.

Returns: Nothing

Usage: Steady State only.

Function Groups: Graphics

Related to:

Format: \System\ProgressBar(LeftReference, BottomReference, RightReference, TopReference, Value[, Low, High, LeftLabel, RightLabel, Align, Delay, ContinuousLoop)

Parameters:

LeftReference

A constant number that gives the left side reference coordinate.

BottomReference

A constant number that gives the bottom side reference coordinate. The top and bottom references are measured down from the top of the screen.

RightReference

A constant number that gives the right side reference coordinate.

TopReference

A constant number that gives the top side reference coordinate.

Value

Required. The value to be displayed as the progress.

Low

Optional numeric, specifying the minimum value for the progress bar. Defaults to zero.

High

Optional numeric, specifying the maximum value for the progress bar. Defaults to 100.

LeftLabel

Optional text providing the left-justified label.

RightLabel

Optional text providing the right-justified label.

Align

Optional numeric. If 0 (the default if unspecified) the graphic will be top-justified. If 1, the graphic will be center-justified.

Delay

Optional numeric. The number of seconds to wait before the graphic is displayed. Defaults to zero.

ContinuousLoop

Optional Boolean. When set true, the progress bar will loop continuously rather than representing the value.

Comments: None

PrtScrn

Description: Prints the image in a window on the default Windows™ printer and returns an error code.

Returns: Numeric

Usage: Script Only.

Function Groups: Printer

Related to: Print | PrintLine | Redirect

Format: PrtScrn([Window, PaletteChanges, PrinterSpec])

Parameters:

Window

An optional object value of a module instance that specifies the window to print. If it is not specified, it defaults to Self which will print the window from which the PrtScrn function is executed.

PaletteChanges

An optional two-dimensional array of RGB values. Each RGB value overrides the default printed color for the palette index, which is the same as the first array subscript.

The first array subscript may have any size (e.g. [0..255] for the whole palette, or [123..125] for just palette indices 123 to 125). (Please see the example in the "Example" section.)

The second array subscript should be sized [0..2] and holds the red, green, and blue components of the color respectively. Each color value must be a real number between zero and one. "0" represents a total absence of that color, while "1" represents the full intensity of the color.

Note: Colors that are in other indexed positions of the palette, but which have the same RGB value as the indexed color to be changed, will usually also change. The danger is that the process of mapping colors from VTS's palette to that of the printer may cause results to differ from those expected. If using the PaletteChanges parameter, you are advised to test this function to determine the actual results.

PrinterSpec

An optional parameter that will accept any of the following:

- Local Printer:

- Port name (including virtual ports) with or without a trailing colon (e.g. DEF or DEF:. COM1 or COM1:; USB001 or USB001:; etc.)
- Windows printer share (e.g. "XYZ Laser Printer")
- Windows share name (if the printer is shared) (e.g. "XYZLaser")
- Local or Remote Printer:
- UNC share name (which includes the host and share name (e.g. "\\localhost\XYlaser" or "\\lab1\NetPrinter")

A note to OEM programmers:
When the VTScada Display Manager or Trend Manager performs a screen print, printer palette color substitution is performed by calling module "GetPrinterPalette". This module is searched for in \Code. The VTS layer does not provide this module. The default printing palette is the display palette. If you wish to provide a separate palette, provide a GetPrinterPalette module. This should be a subroutine module that returns an array of palette color substitution values in the same format required by the PrtScrn statement's PaletteChanges parameter.

Comments:

The return value for this function is as follows

Return Value	Description
0	Success
1	Printing
2	Failure

When printing using this function, make sure that the print job is spooled to the Windows™ Print Manager, otherwise execution of the application will be suspended for the duration of the print job.

Care should be taken to ensure that the graphics in the window you are printing do not change while printer output is being generated. If the graphics do change, the printer output is unpredictable.
In VTS 7.0 and later, all print functions are now compatible with the values returned in either of the first two parameters of the new PrintDialogBox function.

Note: Within an Anywhere Client session, this function does nothing.

Example:

```
If MatchKeys(2, "p") && ! wait;
[
  wait = PrtScrn();
]
```

This will cause the window containing the PrtScrn statement to be printed out when the "P" key is pressed so long as another print job is not already printing or has failed.
Another example might be:

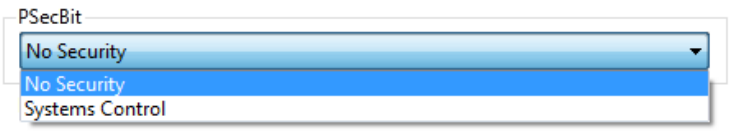
```
PalValues = New(256, 3); { Sized enough for entire palette }
PalValues[0][0] = 1;  { Red - Full intensity }
PalValues[0][1] = 1;  { Green - Full intensity }
PalValues[0][2] = 1;  { Blue - Full intensity }
.
.
.
PrtScrn(Self(), PalValues);
```

This will set the color for palette index 0 (default black) to be white.

PSecBit

(Dialog Library)

Description: Parameter Setting Security Bit. This module draws a titled, [beveled] droplist of options for setting the security bit.



Returns: Nothing

Usage: Steady State only.

Function Groups: Bitwise Operation, Graphics, Security

Related to: GUITransform | PAddressEntry | PAreaSelect | PCheckBox | PColorSelect | PContributor | PDroplist | PEditfield | PPageSelect | PRadioButtons | SelectGraphic | PSpinbox | PTypeToggle

Format: \DialogLibrary\PSecBit(ParmNum, Title, FocusID[, VertAlign, AlignTitle, DrawBevel])

Parameters:

ParmNum

Required. Any numeric expression giving the parameter number (from 0) in the caller to alter.

Title

Required. Any text expression to be used as a title for the droplist.

FocusID

Boolean. If this value is FALSE (0), the field will display its current setting, but cannot be opened (i.e. its value cannot be changed), and will appear disabled (grayed-out).

VertAlign

An optional parameter that is any numeric expression

Value	Vertical Alignment
0	Top
1	Center
2	Bottom

Whether or not the title is included when the vertical alignment is calculated is determined by the value of AlignTitle. The default value is 0.

AlignTitle

An optional parameter that is any logical expression. If AlignTitle is true (non-0), the title will be included in the calculation for vertical alignment.
If AlignTitle is false (0), the title will be added to the editfield after both the editfield and its bevel have been vertically aligned. The default is true.

DrawBevel

An optional parameter that is any logical expression. if true, a bevelled edge will be added around the control.

Comments

This module is a member of the VTScada Dialog Library and must therefore be called from within a GUITransform and prefaced by \DialogLibrary\.
This parameter tool expects the first parameter of its calling module to contain an array of tag parameters. It will then set the element indicated by ParmNum to the chosen security bit.
The height of the (unopened) droplist is constant, with the horizontal boundaries of its calling transform defining its width, and the vertical boundaries of its calling transform defining its opened height, which will include the added height of the bevel above the field, but may or may not include the title, depending on the alignment used. Note that if the entire list can be displayed in a smaller area than indicated by the vertical boundaries of the calling transform, the dropped list height will be decreased. The dropped height of the list will always have a minimum height of 1 line (below the

field).
For any optional parameter that is to be set, all optional parameters preceding the desired one must be present, although they may be invalid.
Usual height: 45 pixels.

Example:

```
GUITransform(50, 400, 450, 100,
             1, 1, 1, 1, 1,
             0, 0, 1, 0,
             0, 0, 0,
             \DialogLibrary\PSecBit(9 { Parm num },
             "Security Bit" { Title },
             3 { Focus ID },
             1 { Centered },
             0 { Align bevel }));
```

PSelectObject

(Dialog Library)

Description: Parameter Setting Select Tag Object Tool. This module draws a beveled, titled droplist of existing tags of a certain type and a new tag creation button.

Returns: Nothing

Usage: Steady State only.

Function Groups: Graphics

Related to: GUITransform | PAddressEntry | PAreaSelect | PCheckBox | PColorSelect | PContributor | PDroplist | PEditfield | PPageSelect | PRadioButtons | PSecBit | PSpinbox | PTypeToggle

Format: \DialogLibrary\PSelectObject(ParmNum, TagType [, Title, FocusID, VertAlign, AlignTitle, DisableDeselect, DrawBevel, AllowRootSelection])

Parameters:

ParmNum

Required. Any numeric expression giving the parameter number (from 0) in the caller to alter.

TagType

Required. May be a text expression for the type or type-group of tag to be used to create the droplist. (Type-groups include "ports", "drivers", "analogs", etc.) Alternatively, this may be an array of types. The array may not include groups.
All existing tags of the type or types will be listed.

Title

An optional parameter that is any text expression to be used as a title for the droplist. If not provided, the title will be blank.

FocusID

Boolean. If this value is FALSE (0), the field will display its current setting, but cannot be opened (i.e. its value cannot be changed), and will appear disabled (grayed-out).

VertAlign

An optional parameter that is any numeric expression that sets the vertical alignment of the unopened droplist according to one of the following:

VertAlign	Vertical Alignment
0	Top
1	Center
2	Bottom

Whether or not the title is included when the vertical alignment is calculated is determined by the value of AlignTitle. The default value is 0.

AlignTitle

An optional parameter that is any logical expression. If AlignTitle is true (non-0), the title will be included in the calculation for vertical alignment.
If AlignTitle is false (0), the title will be added to the editfield after both the editfield and its bevel have been vertically aligned. The default is true.

DisableDeselect

Optional Boolean. Set TRUE to disable the Deselect button

DrawBevel

Optional Boolean. Set FALSE to disable the bevel. Defaults to TRUE.

AllowRootSelection

Optional Boolean. Set TRUE to allow the Root tag to be selected. Default is FALSE

Comments:

This module is a member of the VTScada Dialog Library and must therefore be called from within a GUITransform and prefaced by \DialogLibrary\.
This parameter tool expects the first parameter of its calling module to contain an array of tag parameters. It will then set the element indicated by ParmNum to the chosen item.
The height of the (unopened) droplist and the create new tag button are constant, with the horizontal boundaries of the calling transform defining their combined width, and the vertical boundaries of the calling transform defining the droplist's opened height, which will include the added height of the bevel above the field, but may or may not include the title, depending on the alignment used. Note that if the entire list can be displayed in a smaller area than indicated by the vertical boundaries of the

calling transform, the dropped list height will be decreased. The dropped height of the list will always have a minimum height of 1 line (below the field).
For any optional parameter that is to be set, all optional parameters preceding the desired one must be present, although they may be invalid.
Usual height: 45 pixels.

Example:

```
GUITransform(170, 160, 370, 60,
             1, 1, 1, 1, 1,
             0, 0, 1, 0,
             0, 0, 0,
             \DialogLibrary\PSelectObject(2 { Parm num },
             "AnalogInput" { Tag type },
             "Analog Inputs" { Title },
             2 { Focus ID },
             1 { Center list },
             0 { Align top of bevel }));
```

```
IF Watch(1);
[
  TagTypes = New(3);
  TagTypes[0] = "Calculation";
  TagTypes[1] = "AnalogInput";
  TagTypes[2] = "AnalogStatus";
  TagTypes[3] = "AnalogOutput";
  TagTypes[4] = "AnalogControl";
]
GUITransform(170, 160, 370, 60,
             1, 1, 1, 1, 1,
             0, 0, 1, 0,
             0, 0, 0,
             \DialogLibrary\PSelectObject(2 { Parm num },
             TagTypes { Tag type },
             "Analog Ins, Outs & Calculations" { Title },
             2 { Focus ID },
             1 { Center list },
             0 { Align top of bevel }));
```

PSpinbox

(Dialog Library)

Description: Parameter Setting Spinbox. This module draws a spinbox with optional label.

Returns: Nothing

Usage: Steady State only.

Function Groups: Graphics

Related to: GUITransform | PAddressEntry | PAreaSelect | PCheckBox | PColorSelect | PContributor | PDroplist | PEditfield | PPageSelect | PRadioButtons | PSecBit | PSelectObject | PTypeToggle

Format: \DialogLibrary\PSpinBox(ParmNum, Label, BoxOnLeft, LowLimit, HighLimit [, Alignment, NumChars, CanEdit, FocusID, TextOption, TextValue, Trigger])

Parameters:

ParmNum

Required. Any numeric expression giving the parameter number (from 0) in the caller to alter.

Label

Required. Any text expression to be used as a label with the spinbox.

BoxOnLeft

Required. Any logical expression. If true (non-0) the spinbox will appear to the left of the label, if false (0) it will be to the right. Defaults to TRUE.

LowLimit

Required. Any numeric expression giving the lowest permissible value. If the spinbox is editable and a value less than LowLimit is entered, it will revert to the LowLimit value.

HighLimit

Required. Any numeric expression giving the highest permissible value. If the spinbox is editable and a value

greater than HighLimit is entered, it will revert to the HighLimit value.

Alignment

An optional parameter that is any numeric expression that sets the alignment of the spinbox and its label according to one of the following options. The default value is 0.

Alignment	Horizontal Alignment	Vertical Alignment
0	Left	Top
1	Right	Top
2	Full	Top
3	Left	Centered
4	Right	Centered
5	Full	Centered
6	Left	Bottom
7	Right	Bottom
8	Full	Bottom

NumChars

An optional parameter that is any numeric expression giving the number of digits wide to make the spinbox. A value of 0 or invalid results in the spinbox being automatically sized to fit the widest number (or text string if TextOption and TextValue are set). The default value is 0.

CanEdit

An optional parameter that is any logical expression. If true (non-0), the number in the field may be edited directly, if false (0), it may not. The default value is false. Note that the value of this parameter directly affects

the TextOption and TextValue parameters' effectiveness. If CanEdit is true, both are ignored.

FocusID

Boolean. If this value is FALSE (0), the field will display its current setting, but cannot be opened (i.e. its value cannot be changed), and will appear disabled (grayed-out).

TextOption

An optional parameter that is any text expression used to replace a certain value (expressed by TextValue) in the spinbox field. This parameter will be ignored if CanEdit is true.

TextValue

An optional parameter that is any numeric expression for the value in the spinbox that is to be replaced by the text string in TextOption. This parameter will be ignored if CanEdit is true.

Trigger

An optional numeric expression. The value in Trigger will become 0 if the user changes the internal buffer (i.e. when the value of the WinEditCtrl as logged in the variable Change transits from invalid to zero). If the user presses any of Enter, the spin box arrows or the arrow buttons on the keyboard, Trigger becomes 1. If the spinbox loses focus, the value of Trigger becomes 2.

Comments: This module is a member of the VTScada Dialog Library and must therefore be called from within a GUITransform and prefaced by \DialogLibrary\. This parameter tool expects the first parameter of its calling module to contain an array of tag parameters. It will then set the element indicated by

ParmNum to the value displayed in the spinbox.
The size of the spinbox is constant, with the boundaries of the calling transform defining the position of the check box and its label.
For any optional parameter that is to be set, all optional parameters preceding the desired one must be present, although they may be invalid.
Usual height: 22 pixels.

Example:

```
GUITransform(30, 100, 130, 20,
             1, 1, 1, 1, 1,
             0, 0, 1, 0,
             0, 0, 0,
             \DialogLibrary\PSpinbox(4 { Parm num },
               "Number of retries" { Label },
               1 { Box on left },
               0, 100 { Limits },
               5 { Full align, centered },
               0 { Autosize box },
               1 { Editable },
               1 { Focus ID },
               "None" { Text option },
               0 { Index of text option },
               Trigger {Trigger} ));
```

PType

Description: Returns the actual type of parameter at an index.

Returns: Numeric – see comments

Usage: Script Only.

Function Groups: Advanced Module, Variable

Related to: NParm | Parameter | ResetParm

Format: PType(Object, Index)

Parameters:

Object

Required. Any object (the object value of a running

module instance).

Index

Required. Any numeric expression giving the number of the parameter of interest, starting from 1.

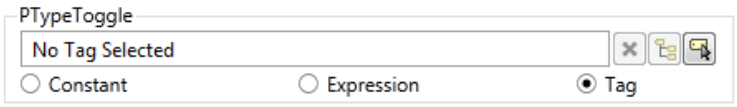
Comments: This function is for experienced users, and is not needed for normal operation. The return value reveals the type of parameter passed to the module from the module call, and is determined by the following table

Value	Meaning
0	Status reference
1	Short reference
2	Long reference
3	Float reference
4	Text reference
5	Object reference
6	Status value (formal parameter type)
7	Short value (formal parameter type)
8	Long value (formal parameter type)
9	Float value (formal parameter type)
10	Text value (formal parameter type)
11	Object value (formal parameter type)
12	Calculated value (not a reference or constant)
13	Short constant without formal parameter
14	Long constant without formal parameter
15	Float constant without formal parameter
16	Text constant without formal parameter

PTypeToggle

(VTS Library)

Description: Parameter Setting Type Toggled Field. This module draws a beveled droplist or editfield with title that sets a tag or numeric value.



Returns: Nothing

Usage: Steady State only.

Function Groups: Graphics, Variable

Related to: GUITransform | PAddressEntry | PAreaSelect | PCheckBox | PColorSelect | PContributor | PDroplist | PEditfield | PPageSelect | PRadioButtons | PSecBit | PSelectObject | PSpinbox

Format: \DialogLibrary\PTypeToggle(ParmNum, TagType, Title, FocusID, VertAlign, AlignTitle[, LowLimit, HighLimit, Trigger, EnableExpressions, DrawBevel, PtrValue, Immutable, AllowRootSelection, ParmValueType, StatusLabel])

Parameters:

ParmNum

Required. Any numeric expression giving the parameter number (from 0) in the caller to alter.

TagType

Required. Any text expression for the type of tag used to create the droplist portion of the graphic. All existing tags of this type will be listed.

Title

An optional parameter that is any text expression to be used as a title for the tool. May be left as Invalid for no title.

FocusID

Boolean. If this value is FALSE (0), the field will display its current setting, but cannot be opened (i.e. its value cannot be changed), and will appear disabled (grayed-out).

VertAlign

Required. Any numeric expression that sets the vertical alignment of the unopened droplist according to one of the following:

VertAlign	Vertical Alignment
0	Top
1	Center
2	Bottom

Whether or not the title is included when the vertical alignment is calculated is determined by the value of AlignTitle. The default value is 0.

AlignTitle

Required Boolean. If AlignTitle is true (non-0), the title will be included in the calculation for vertical alignment.

If AlignTitle is false (0), the title will be added to the editfield after both the editfield and its bevel have been vertically aligned. The default is true.

LowLimit

An optional parameter that is any numeric expression giving the minimum value to be accepted by the editfield portion of the tool (it does not affect the tag's value if a tag is accepted).

If this parameter is valid and a value less than LowLimit is entered in the field, the variable set by the field will revert to the previous value.

HighLimit

An optional parameter that is any numeric expression giving the maximum value to be accepted by the edit-field portion of the tool (it does not affect the tag's value if a tag is accepted).
If this parameter is valid and a value greater than HighLimit is entered in the field, the variable set by the field will revert to the previous value.

Trigger

An optional parameter that is set when the variable is changed.

EnableExpressions

An optional parameter that indicates the initial state of DoEnableExpressions.
If EnableExpressions is set to 1, a radio button will be shown that toggles display of the ExpressionEdit widget, which in turn allows users to enter an expression. EnableExpressions defaults to 0, but if the value controlled by the PTypeToggle is already an expression, then the expressions option will be displayed regardless.

DrawBevel

TRUE to draw Bevel

PtrValue

Used to return the value of expressions/tags

Immutable

TRUE if we want to force all tag references to generate immutable reference code. Defaults to FALSE.

AllowRootSelection

TRUE to allow the Root tag to be selected. Default is FALSE

ParmValueType

ValueType of the parameter:

0 = Status (boolean)

1 = Short

2 = Long

3 = Double

4 = Text

5 = Octal

6 = Hexadecimal

7 = Color

StatusLabel

Label for the type Status radio button

Comments:

This module is a member of the VTScada Dialog Library and must therefore be called from within a GUITransform and prefaced by \DialogLibrary\.
This parameter tool expects the first parameter of its calling module to contain an array of tag parameters. It will then set the element indicated by ParmNum to the numeric value or the name of the selected tag.
The height of the (unopened) droplist, the create new tag button and the radio buttons beneath them are constant, with the horizontal boundaries of the calling transform defining the combined width of the droplist and button, and the vertical boundaries of the calling transform defining the droplist's opened height, which will include the added height of the bevel above the field, but may or may not include the title, depending on the alignment used. Note that if the entire list can be displayed in a smaller area than indicated by the vertical boundaries of the calling transform, the dropped list height will be decreased. The dropped height of the list will always

have a minimum height of 1 line (below the field). For any optional parameter that is to be set, all optional parameters preceding the desired one must be present, although they may be invalid. This control is not to be used in Panel modules as it creates undesired output and does not support the range of type selections required for widget configuration. In panel modules, use ParameterEdit or NumericParameterEdit instead.
Usual height: 55-100 pixels.

Example:

```
GUITransform(10, 160, 270, 60,
             1, 1, 1, 1, 1,
             0, 0, 1, 0,
             0, 0, 0,
             \DialogLibrary\PTypeToggle(1 { Parm num },
             "DigitalOutput" { Tag type },
             "Choose tag" { Title },
             1 { ID },
             0 { Top align list },
             1 { Align title },
             10, 20 { Limits for the value }));
```

Q Functions

The sections that follow identify all VTScada functions beginning with "Q".

QuietLogon

Security Manager Module

Description: Authenticates the AuthToken and, if successful logs the calling user session on as the user specified in the AuthToken

Returns: Boolean

Usage: Script Only.

Related to: AlternateIdCheck | AlternateLogoff | AlternateLogon | Authenticate | LogOff | UserCredChange | UserLogonDialog

Format: \SecurityManager\QuietLogon(AuthToken [, Device, Namespace]);

Parameters:

AuthToken

The concatenation of the user name, a colon (:) and the password.

Device

Optional. Name of the device that is making the request. Defaults to none.

Namespace

Optional. The namespace of the user. Defaults to none.

Comments: If the authentication fails, a failure event is recorded in the security event log.
Returns TRUE if the user was logged on successfully, otherwise FALSE.

R Functions

The sections that follow identify all VTScada functions beginning with "R".

RadialIndicator

(Meter Parts Library)

Description: Will draw a radial type indicator that sweeps from a minimum angle to a maximum angle in the same fashion that a real radial meter would. Call from within a GUITransform.

Returns: Nothing

Usage: Steady State only.

Function Groups: Graphics

Related to: GUITransform | RadialLegend

Format: \MeterParts\RadialIndicator(DataSource, IndicatorImage[, OffsetFromCenter, Direction, MinimumAngle, MaximumAngle, Hue, Saturation, Brightness, Transparency, Contrast, ColorizeHue, ColorizeIntensity, UseTagScaling, MinScaleValue, MaxScaleValue, DampenIndicator])

Parameters:

DataSource

Required. A Tag name, constant or expression that represents the value to show.

IndicatorImage

Required. The full path to the name of an image file to use as the indicator. Typically, this is an image of a needle.

OffsetFromCenter

The offset from the center to translate the image. This allows the user to change the rotation center point of the indicator image. The default is 0 which sets the bottom of the indicator image as the rotation center point.

Direction

A flag indicating the direction from the minimum to the maximum value representation. 0 means increase in a clockwise manner while 1 indicates a counter-clockwise manner. The default is 0 (clockwise).

MinimumAngle

The start angle of the sweep. 0 is defined as up or the 12 o'clock position. The default is 225.

MaximumAngle

The end angle of the sweep. 0 is defined as up or the

12 o'clock position. The default is 135.

Hue

The Hue translation to perform on the Indicator image. This enables you to change the color of the indicator image. The image must have color in it already in order to perform a hue translation. If there is no color to start with, then changing this value does nothing. You can add color by setting a value for the ColorizeHue parameter, described later.
The default is 0, indicating that no hue translation is done and the indicator is in its native color.

Saturation

The amount of saturation of the colors in the indicator image. A value of 0 will make the image black and white (no color saturation). A value of 2 produces a brightly colored (saturated) indicator. The default is 1 which corresponds to the native saturation of the indicator image.

Brightness

An adjustment of the brightness of the indicator image. Higher numbers produce a brighter indicator image. A 0 produces a black image. The default is 1 which corresponds to the native brightness of the indicator image.

Transparency

An adjustment of the opacity of the indicator where 1 means 100% opacity and 0 means %100 transparent. The default is 1.

Contrast

An adjustment of the contrast of the colors in the indicator image. A value of 0 produces a flat looking image and a value of 2 gives a high contrast image. The default is 1 which corresponds to the native contrast of

the indicator image.

ColorizeHue

A value that works in conjunction with ColorizeIntensity. This is the hue of the color that is introduced by colorizing an image. Colorizing an image will introduce color into an image that previously was black and white or grayscale. The default value is 0.

ColorizeIntensity

A value to define how much color to introduce into the image. The default is 0, meaning not to introduce any color at all into the image.

UseTagScaling

A flag that indicates whether or not to use the supplied Tag's scaling values. The default is FALSE.

MinScaleValue

The minimum scale value to use if the UseTagScaling flag is not true. The default is 0.

MaxScaleValue

The maximum scale value to use if the UseTagScaling flag is not true. The default is 100.

DampenIndicator

A flag to indicate whether or not to dampen the indicator movement. Dampened movement creates the effect of animating the indicator. The default is false.

Comments: This function must be called within a GUITransform statement in order for it to work correctly.
The size of the indicator is scaled with respect to the original size of the image and the size of the transform. If you want a smaller indicator you can simply make a smaller transform. Using the offset from center in conjunction with the size of the transform allows for extensive customization of the size and position of the needle.

Example:

```
GUITransform(458, 392, 608, 242,
             1, 1, 1, 1, 1 { Scaling             },
             0, 0          { Movement            },
             1, 0          { Visibility, Reserved },
             0, 0, 0       { Selectability       },
             \MeterParts\RadialIndicator(Invalid,
             "Bitmaps\Meter Parts\Indicators\Radial\Needle5.png",
              0, 0, 225, 135, 0, 1, 1, 1, 1, 0, 0, 0, 0, 100, 0));
```

RadialLegend

(Meter Parts Library)

Description: Draws a legend (i.e. the text labels) for a radial type meter. They are drawn at a constant radius from the center point of the drawing coordinates, beginning at a defined minimum angle and ending at a defined maximum angle. To be called from within a GUITransform.

Returns: Nothing

Usage: Steady State only.

Function Groups: Graphics

Related to: GUITransform | RadialLegend

Format: \MeterParts\RadialLegend(TagName[, MinimumAngle, MaximumAngle, NumLabels, Font, Color, Direction, Orientation, UseTagScaling, MinScaleValue, MaxScaleValue])

Parameters:

TagName

Required. The name of the Tag to use for scaling. If no tag is specified, then tag scaling cannot be used to automatically obtain the minimum and maximum scale values.

MinimumAngle

The start angle of the sweep. 0 is defined as up or the 12 o'clock position. The default is 225.

MaximumAngle

The end angle of the sweep. 0 is defined as up or the 12 o'clock position. The default is 135.

NumLabels

The number of Labels to show. The default is 5.

Font

The name of a font tag to use for the legend text.

Color

A color index for the color of the legend text. The default is 0 (black).

Direction

A flag indicating the direction from the minimum to the maximum value representation. 0 means increase in a clockwise manner while 1 indicates a counter-clockwise manner. The default is 0 (clockwise).

Orientation

A reserved parameter and should be set to 0.

UseTagScaling

A flag that indicates whether or not to use the supplied Tag's scaling values. The default is false.

MinScaleValue

The minimum scale value to use if the UseTagScaling flag is not true. The default is 0.

MaxScaleValue

The maximum scale value to use if the UseTagScaling flag is not true. The default is 100.

Comments:

This function must be called within a GUITransform statement in order for it to work correctly.
The text should scale with the size of the transform, if it does not, then you might have picked a font that doesn't scale. Some non true-type fonts won't scale.

Example:

```
GUITransform(478, 616, 628, 466,
             1, 1, 1, 1, 1 { Scaling             },
             0, 0          { Movement            },
             1, 0          { Visibility, Reserved },
             0, 0, 0       { Selectability       },
             \MeterParts\RadialLegend(Invalid, 225, 135, 5, Invalid,
                                      0, 0, 0, 0, 0, 100));
```

RadioButtons

(System Library)

Description: Draws a set of labeled radio buttons with (optional) title and border.

Returns: Nothing

Usage: Steady State only.

Function Groups: Graphics

Related to: CheckBox | Droplist | Listbox | Spinbox | SplitList | ToolBar | VScrollbar

Format: \System\RadioButtons(X1, Y1, X2, Y2, Labels, Select [, FocusID, Border, Title, BtnsOnLeft, AlignTitle, AllowTextResizing, LeftToRight, BGColor, FGColor, Spacing])

Parameters:

X1

Required. Any numeric expression giving the X coordinate on the screen of one side of the radio buttons.

Y1

Required. Any numeric expression giving the Y coordinate on the screen of either the top or bottom of the radio buttons.

X2

Required. Any numeric expression giving the X coordinate on the screen of the side of the radio buttons opposite to X1.

Y2

Required. Any numeric expression giving the Y coordinate on the screen of the top or bottom of the radio buttons, whichever is the opposite to Y1.

Labels

Required. An array of text expressions used to label the buttons. The number of labels determines the number of buttons.

Select

Required. A variable whose value will be set to the index from 0 of the selected button. If the value of this variable is initially invalid, it will default to 0.

FocusID

An optional parameter that is any numeric expression for the focus number of the first radio button. Each radio button following will have a focus ID number equal to the next number in sequence.
If this value is 0, the radio buttons will display the current setting of Select, but its value will not be able to be changed and the buttons will appear grayed out. The default value is 1.

Border

An optional parameter that is any logical expression. If true (non-0) the buttons will have a background outline, if false (0) there will be no outline around the buttons. The default is true.

Title

An optional parameter that is any text expression to be used as a title with the radio buttons. The default value is to have no title.

BtnsOnLeft

An optional parameter that is any logical expression. If true (non-0) the radio buttons will appear to the left of their labels, if false (0) they will be to the right. The

default value is true.

AlignTitle

An optional parameter that is any logical expression. If true (non-0) the title is drawn within the radio buttons' boundaries, if false(0) the buttons fill their bounding area and the title is added at the top (i.e. it extends past the top boundary). The default is true.

AllowTextResizing

An optional flag. If true, then text will resize with the control.

LeftToRight

An optional flag. If true, the buttons will be drawn from left to right.

BGColor

Optional. Any numeric expression for the background color of the control. No default value.

FGColor

Optional. Any numeric expression for the foreground color of the control (text color). No default value.

Spacing

Optional. An array of numeric values, providing the spacing in pixels to be used between one button and the next.

Comments:

This module is a member of the System Library, and must therefore be prefaced by \System\, as shown in the "Format" section. If you are developing a script application, use "System\..." rather than "\System\..." in the function call.

The parameters X1, Y1, X2 and Y2 define the outline of the buttons including their border, if there is one. If the area is too small to fully display the buttons they will extend beyond their right and bottom boundaries. Buttons and their

border will not overlap each other and will always be shown in their entirety, although the labels may be clipped or entirely deleted.
For any optional parameter that is to be set, all optional parameters preceding the desired one must be present, although they may be invalid.

Example:

```
System\RadioButtons(10, 10, 300, 210 { Outline of buttons },
                    Types { An array of labels },
                    Selected { Selected btn (from 0)},
                    4 { First btn's focus ID },
                    1 { Show border },
                    "PLC Types" { Title of the buttons },
                    0 { Buttons on the right });
```

Rand

Description	Returns a random number between 0 and 1.
Returns	Numeric
Usage	Script only. May be used in optimized Tag Parameter Expressions.
Function Groups	Generic Math
Related to:	Scale
Format:	Rand()
Parameters	None
Comments	This function is useful for simulations.

Example:

```
If TimeOut(1, 2);
[
   simulatedTemperature = Scale(Rand(), 0, 1, 50, 150);
]
```

Every 2 seconds, this sets simulatedTemperature to a random value between 50 and 150.

Read

(VTSDriver Library)

Description: Used by a tag to create a request for a single read of a given driver address. (This is in contrast to the polled read request of the AddRead function).

Returns: Object value of underlying read module.

Usage: Script Only.

Function Groups: Stream and Socket

Related to: AddRead

Format: …\Driver\Read(Address, N, PtrDataDest[, OriginalAddr])

Parameters:

Address

Required. The starting address of the data to be read.

N

Required. The number of elements to read.

PtrDataDest

Required. A pointer to destination for the data. May be a pointer to a variable, an array or the object value of a module. Will be set to Invalid if the read fails.

OriginalAddr

The original address string from an I/O tag. May be different from the address for a DriverMUX.
If provided, this will become the first parameter to a NewData call.

Comments: Allows the reading of a specific address on demand. The resulting data will be sent only to the requesting machine. This will also be the case when the function is run in a client-of-a-client configuration. The object value of the underlying read module is

returned from the function. When the read finishes, the returned object's value will go to Invalid, signaling the end of read.
This function does not support bit extraction or type conversions. Use BuffRead on the result, followed by BuffWrite to do a type conversion. Depending on how the float was formed by the PLC a call to BuffOrder might be needed to arrange the bytes correctly. To perform bit extraction, use the Bit function on the result of the Read.

Example:

```
If 1 WaitForReadToEnd;
[
  Obj = ModDrv1\Driver\Read(40001, 5, &Val);
]

WaitForReadToEnd [
  If !Valid(Obj);
  [
    { … process data from read … }
    { When Obj is Invalid, read operation is complete and Val    }
    { will be an array of 5 values.                              }
    { If the read did not succeed, such as in the case of server }
    { failover, Val would be an array of 5 Invalid elements.  }
  ]
]
```

This example reads from addresses 40001 to 40005 of a Modbus driver tag named ModDrv1 and returns the result as an array in the variable Val.

Related Information:

Bit, BuffRead, BuffWrite, BuffOrder

ReadBlock

(VTSDriver Library)

Description Is launched to read a block of data from the PLC. It maintains a linked list of pointers to tag values with their absolute offset into the PLC file being read by this instance.

Warning This function is for use by advanced programmers only. It is used only as a part of the VTSDriver module. Engineers writing a driver will not need to call this function directly.

Returns Linked List

Usage Script Only.

Function Groups Stream and Socket

Related to:

Format: …\Driver\ReadBlock(Info1, Info2, Info3, DType)

Parameters

Info1

Driver dependant. The first grouping parameter.

Info2

The second grouping parameter.

Info3

The third grouping parameter.

DType

The data type to read from the I/O device.

Comments Launched from a script, runs in steady state.
This module itself is in a global linked list of instances maintained by the VTSDriver module. This block represents all of the data to be read from a specific file in a specified PLC. All scan rates are handled by this one instance. This module keeps a list of nodes sorted by scan rates. These nodes point to another list of all the memory requests within that scan rate. This module will determine the best organization of blocks and launch a separate read module for each actual block of data read.

ReadConfiguration

Description: This function provides a safe way to read configuration

files.

Returns: Nothing

Usage: Script Only.

Function Groups: File I/O

Related to: ModifyConfiguration | ReadPropertiesFile | WCSubscribe

Format: \ReadConfiguration(CallBackModuleName)

Parameters:

CallBackModuleName

Required text value, which is the name of the module (either launched or a subroutine that returns Invalid) . The named module is launched in the caller.

Comments: The callback object is allowed to read configuration files and is guaranteed that, for the life of the module, no other configuration code can modify the file. The callback module need not have parameters of its own. ReadConfiguration will not put values in them if they exist.

Example:

```
  If SomeTrigger;
  [
    SomeTrigger = 0;
    \ReadConfiguration("ConfigReader");
  ]
…
]

<
{============================ \ConfigReader
=========================}
{=========================================================================-
==}
ConfigReader

Main [
  If 1;
  [
     { The first line of the file is read into InfoVal1, and the
second }
     { line is read into InfoVal2.
```

```
}
    FRead("MyConfigFile.txt" { filename  },
          0                  { offset    },
          "%l%l"             { format string   },
         InfoVal1, InfoVal2  { return variables });
    Return(Invalid);
  ]
]
{ End of \ConfigReader }
>
```

ReadINI

(System Library)

Description: This subroutine read a variable entry from an INI file or a buffer containing one and returns its value. Will not access .Startup or .Dynamic files.

Note: Access to configuration files is not reliable unless the caller holds the working copy lock. Acquiring the lock is a steady-state only operation, and therefore legacy operations that used script-mode access to these files are deprecated or no longer supported (see comments)

Returns: Varies

Usage: Script only.
May be used in optimized Tag Parameter Expressions.

Function Groups: File I/O

Related to: ReadConfiguration | CheckFileExist | CheckPathExist | ReadSectINI | WriteINI | WriteSectINI |

Format: \System\ReadINI(File, Section, VarName [, UseBuff])

Parameters:

File

Required. Any text expression giving the absolute path and file name of the configuration file or the actual buffer containing its contents, depending on the UseBuff parameter.

Section

Required. Any text expression giving the name of the section in the file. This should not include the square brackets delimiting the section.

VarName

Required. Any text expression giving the name of the variable for which the value is required.

UseBuff

An optional parameter that is any logical expression. If true (non-0) the value of File must be a buffer, if false (0) it is a file that is to be used. The default used if this parameter is omitted is false.

Comments:

For developers the lock means that access to VTScada working copy files, both reading and writing, should not be done without having the lock. The lock is across all applications and system layer VTScada code. The lock prevents two different piece of code from changing the same code such that one piece of code sees inconsistent data while the other code is in the middle of changing it

This subroutine was a member of the System Library, and must therefore be prefaced by \System\, as shown in the "Format" section. If you are developing a script application, use "System\..." rather than "\System\..." in the function call.

The return value will be invalid if the configuration file, section or variable was not found. Searches performed by this function are case insensitive.

Example:

```
If 1 Main;
[
  DSource = System\ReadINI("C:\VTScada\Setup.INI" { Name of file },
                           "System" { Name of section },
                           "OrderlyShutdown" { Name of variable },
                           0 { Read file format });
]
```

This assigns the value of the variable OrderlyShutdown in the System section of the Setup.INI file to the variable called DSource.

ReadINIProperties

Description: Gathers the sum of all of the properties files in this layer and all of its parents including the local workstation files.

Returns: Dictionary (see comments)

Usage: Script Only.

Function Groups: Configuration Management

Related to: ReadINI | ReadConfiguration | ReadPropertiesFile

Format: Layer\ReadINIProperties(Result[, ExternalLock, SuppressOrphanedComments]);

Parameters:

Result

Required. A pointer to a value that, when all files have been read, is set to the dictionary of structures described in the comments section. If this is already a valid dictionary and the FileName parameter is valid, the data for the single file will be updated in the dictionary. This feature allow the publisher calls to Notifiy to update the structures.

ExternalLock

Optional Boolean. Set to TRUE if you do not want to acquire and release the lock. Defaults to FALSE.

SuppressOrphanedComments

Optional Boolean. Set to TRUE to ignore "+PseudoProperty" - comments that are not associated with a property. Defaults to FALSE.

Optional. Any

Comments: This is a long operation, but is useful for reading properties that could be inherited. The data is returned in a dictionary of structures, as follows. The key is the layer's GUID.The dictionary must be ordered with the application layer first and the base layer last. The Result parameter must be a valid dictionary that is populated by this module.

```
LayerProperties Struct [
  Layer { Instance of application layer owning
the files };
  Files { Dictionary of INIFiles structures for
a layer };
]
```

```
INIFiles Struct [
  FileName { Name (without the path) of the set-
tings file };
  Workstation { Name of the workstation or
invalid if global };
  Dynamic { TRUE if a dynamic property };
  Sections { Dictionary of sections each element
of which is an array of Property structures };
  Changed { User sets to true if the file has
been changed, initialized to false };
]
```

```
INIProperty Struct [
  Name { Variable name in the settings file };
  Value { Simple Value or an ordered array of
Values if the variable occurs more than once in
the section of the file };
  Comment { Text comment if present in the file
};
]
```

A simpler function, ReadPropertiesFile, is much more direct if the location of the setting to be read is known.

Examples:

ReadLock

(RPC Manager Library)

Description: Attempts to acquire a Read lock for the specified service. Subroutine call only.

Returns: Nothing

Usage: Script Only.

Related to: WriteLock

Format: \RPCManager\ReadLock(ActivePtr, Service [, OptGUID]);

Parameters:

ActivePtr

Required. A pointer to a variable that will be set to "1" when the Read lock is obtained.

Service

Required. The name by which the service is known.

OptGUID

An optional parameter that is any expression giving the 16-byte binary form of the globally unique identifier (GUID) for the application in which the service instance is located. The default is the application to which the caller belongs.

Comments: This subroutine is a member of the RPC Manager's Library, and must therefore be prefaced by \RPCManager\, as shown in the "Format" section. If the application you are developing is a script application, the subroutine call must be prefaced by System\RPCManager\, and the System variable must be declared in AppRoot.src.
This module maintains a Read lock on a per service basis. This module is intended to be launched. It will set the value pointed to by ActivePtr to 1 when the lock is obtained. To release the lock, or stop waiting for it, simply stop this module by stopping the caller or explicitly slaying it.

ReadPropertiesFile

(System Library)

Description Reads a single Settings file and returns an INIFile Structure. Replaces ReadINI and ReadSectINI

Returns INIFile Structure. See comments.

Usage Script Only.

Function Groups Configuration Management, File I/O

Related to: WritePropertiesFile | GetINIProperty | ReadConfiguration | SetINIProperty

Format: \System\ReadPropertiesFile(File[, IsBuffer, SuppressOrphanedComments])

Parameters

File

Required. Any text expression giving the full path and file name of the Settings file or the buffer containing its contents, depending on the IsBuffer parameter.

IsBuffer

An optional logical expression. Set TRUE if the File parameter is a buffer. Defaults to FALSE (0).

SuppressOrphanedComments

An optional logical expression. If TRUE then "+PseudoProperty"s - comments that aren't associated with a property will be left out. Defaults FALSE.

Comments The INIFile structure returned is as follows:

```
INIFiles Struct [
  FileName    { full path and file name to the
settings file          };
  OEM         { TRUE if an OEM layer file
};
  Workstation { Name of the workstation or
invalid if global           };
  Layer       { Instance of application layer
owning the file          };
  Dynamic     { TRUE if a dynamic property
```

```
};
  Sections    { Dictionary of sections each ele-
ment of which
                is an array of Property struc-
tures        };
  Changed     { User sets to true if the file has
been changed,
                initialized to false
};
 ]
```

The INIProperty structure is…

```
INIProperty Struct [
    Name        { Variable name in the .star-
tup/.dynamic file  };
    Value       { Simple value
};
    Comment     { Text comment if present in the
file          };
    Hidden      { TRUE if not visible in Edit
Properties GUI   };
  ];
```

The INIFiles structure can be modified using SetINIProperty.
Note that if your intention is to read a configuration file, this function should be called from within a ReadConfiguration callback or a ModifyConfiguration callback.

Example:

```
Properties = ReadPropertiesFile(Concat(GetWCPath(),
                                       #APP_INI_FILENAME,
                                       #DYNAMIC_INI_EXT));
Name       = GetINIProperty(Properties\Sections["Application"],
"Name");
```

ReadSectINI

Description: This subroutine read an entire section entry from a configuration file or a buffer containing one and returns a 2-dimensional array containing variable names and their values. Will not access .Startup or .Dynamic files.
Note: Access to configuration files is not reliable unless the caller holds the working copy lock. Acquiring the lock is a

steady-state only operation, and therefore legacy operations that used script-mode access to these files are deprecated or no longer supported (see comments)

Returns: Array

Usage: Script Only.

Function Groups: File I/O

Related to: CheckFileExist | CheckPathExist | ReadINI | WriteINI | WriteSectINI |

Format: \System\ReadSectINI(File, Section [[, UseBuff] , PtrSectionStatus)

Parameters:

File

Required. Any text expression giving the absolute path and file name of the Settings file or the name of the buffer containing its contents.

Section

Required. Any text expression giving the name of the section in the file. This should not include the square brackets delimiting the section.

UseBuff

An optional parameter that is any logical expression. If true (non-0) the value of File must be a pointer to a buffer, if false (0) it is a file that is to be used. The default used if this parameter is omitted is false.

PtrSectionStatus

A flag used to inform the caller of what was found, according to the following table:

Value	Meaning
Invalid	when either of the first 2 parameters are invalid
0	when both section and settings exist
1	when no section is found
2	when section is found, but not settings

Comments:

For developers the lock means that access to VTScada working copy files, both reading and writing, should not be done without having the lock. The lock is across all applications and system layer VTScada code. The lock prevents two different piece of code from changing the same code such that one piece of code sees inconsistent data while the other code is in the middle of changing it.
This subroutine was a member of the System Library, and must therefore be prefaced by \System\, as shown in the "Format" section. If you are developing a script application, use "System\..." rather than "\System\..." in the function call.
The array that is returned gives the variable names in its first row (Array[0][N]) and the variables' values in its second row (Array[1][N]). The return value will be invalid if the Settings file or section was not found, or if the section did not contain any variables. Searches performed by this function are case insensitive.

Example:

```
If 1 Main;
[
  Vars = System\ReadSectINI("C:\VTScada\Setup.ini" { Name of file },
                             "System" { Name of section },
                            0 { Read file format });
]
```

This creates the array Vars and stores in it all variables in the System section of the Settings.Dynamic file.

ReadX

Description Reads numeric data from a text file into the elements of an array.

Returns Array

Usage Script Only.

Function Groups Array, File I/O

Related to: FRead | SRead | ReadXY

Format: ReadX(ArrayElem, N, File)

Parameters

ArrayElem

Required. Any array element giving the starting point in the array in which to store the data. The subscript for the array may be any numeric expression.
If processing a multidimensional array, the usual rules apply to decide which dimension should be used. The array may be either statically declared or dynamically allocated.

N

Required. Any numeric expression giving the number of array elements to read starting at the element given by the first parameter.
If N extends past the upper bound of the lowest array dimension, this computation will "wrap-around" and resume at element 0, until N elements have been processed.

File

Required. A text expression giving the file name of the data file. It should be enclosed in double quotes if it is

a constant. A known path Known Path Aliases for File-Related Functions for File-Related Functions may be provided in the form, :{KnownPathAlias}.

Comments The data should be stored in the file with one numeric value per line. The EOF marker for the file must not be on the same line as the last numeric value, as per the following rule

If more than one value exists on a line, if the line is left blank, or if it contains a non-numeric character, the corresponding element in the array will be set to invalid.
If the file contains fewer than N values, the remaining values will also be invalid.
This statement is useful for importing data from other databases into VTScada.

Example:

```
If MatchKeys(1, "G");
[
  ReadX(chlorineCon[0] { Starting point in array },
       30 { Number of elements to read },
       "CHLORINE.TXT" { File from which to read data });
]
```

This reads 30 numbers from the text file called CHLORINE.TXT into elements 0 to 29 of chlorineCon when a capital G is pressed on the keyboard.

ReadXY

Description: Reads data points from a file into the elements of two arrays.

Returns: Array

Usage: Script Only.

Function Groups: Array, File I/O

Related to: FRead | ReadX | SRead

Format: ReadXY(XArrayElem, YArrayElem, N, File)

Parameters:

XArrayElem

Required. Any array element giving the starting point in the array in which to store the X coordinates. The subscript for the array may be any numeric expression.

If processing a multidimensional array, the usual rules apply to decide which dimension should be used. The array may be either statically declared or dynamically allocated.

YArrayElem

Required. Any array element giving the starting point in the array in which to store the Y coordinates. The subscript for the array may be any numeric expression.

If processing a multidimensional array, the usual rules apply to decide which dimension should be used. The array may be either statically declared or dynamically allocated.

N

Required. Any numeric expression giving the number of array elements to read starting at the elements given by the first two parameters.

If N extends past the upper bound of the lowest array dimension, this computation will "wrap-around" and resume at element 0, until N elements have been processed.

File

Required. A text expression giving the file name of the data file. It should be enclosed in double quotes if it is a constant. A known path Known Path Aliases for File-Related Functions for File-Related Functions may be

provided in the form, :{KnownPathAlias}.

Comments: The data should be stored in the file with one numeric value per line. The data are then alternately placed into each of the two arrays. If more than one value exists on a line, if the line is left blank, or if it contains a non-numeric character, the corresponding element in that array will be set to invalid. If the file contains fewer than N values, the remaining values will also be invalid.

The arrays need not start at the same index number. This statement is useful for importing (X, Y) data from other databases or spreadsheets into VTScada.

Example:

```
If MatchKeys(1, "R");
[
  ReadXY(chlorineCon[0] { Starting point in first array },
        pumpSpeed[0] { Starting point in second array },
        60 { Number of elements to read },
        "CHLORINE.TXT" { File from which to read data });
]
```

This reads 60 numbers from the text file called CHLORINE.TXT into elements 0 to 29 of chlorineCon and pumpSpeed when a capital R is pressed at the keyboard. Numbers are placed in alternating arrays as they are read.

RecommendAlternate

(RPC Manager Library)

Description: Instructs RPC Manager that the local service instance does not consider itself a good server candidate.

Returns: Nothing

Usage: Script Only. (Subroutine call only)

Function Groups: Network

Related to:

Format: \RPCManager\RecommendAlternate(Service [, OptGUID]);

Parameters:

Service

The name by which the service is known.

OptGUID

An optional parameter that is any expression giving the 16-byte binary form of the globally unique identifier (GUID) for the application in which the service instance is located. The default is the application to which the caller belongs.

Comments: This subroutine is a member of the RPC Manager's Library, and must therefore be prefaced by \RPCManager\ as shown in the "Format" section. If the application you are developing is a script application, the subroutine call must be prefaced by System\RPCManager\, and the System variable must be declared in AppRoot.src.

RecommendPrimary

(RPC Manager Library)

Description: Instructs RPC Manager that the local service instance considers itself a good server candidate.

Returns: Nothing

Usage: Script Only. (Subroutine call only)

Function Groups: Network

Related to:

Format: \RPCManager\RecommendPrimary(Service [, OptGUID]);

Parameters:

Service

The name by which the service is known.

OptGUID

An optional parameter that is any expression giving the 16-byte binary form of the globally unique identifier (GUID) for the application in which the service instance is located. The default is the application to which the caller belongs.

Comments: This subroutine is a member of the RPC Manager's Library, and must therefore be prefaced by \RPCManager\, as shown in the "Format" section. If the application you are developing is a script application, the subroutine call must be prefaced by System\RPCManager\, and the System variable must be declared in AppRoot.src.

RecordProperty

Description: Helper function used to record settings without needing to explicitly interact with the settings files. Can modify one or more properties within a single specified file, and commits the result.

Returns: Object (which becomes invalid on completion)

Usage: Script Only.

Function Groups: Configuration Management

Related to:

Format: LayerRoot\RecordProperty(SettingsFile, Section, Name, Value, Comment, CommitComment[, Deploy, CallerHasLock, ClearSection, Workstation])

Parameters:

SettingsFile

Required. Text or Integer. The name of the settings file to be altered.
As a convenience this parameter can also be an integer representing any of the four standard settings file types as follows:
0 => workstation.Dynamic

1 => Settings.Dynamic

2 => workstation.Startup

3 => Settings.Startup

Section

Required. May be the name of the section that the property belongs to, or an array of section names.

Name

Required. May be the name of the property to modify, or an array of property names.

Value

Required. May be the value to be set for the property, or an array of values matching the array of property names.

Comment

Optional. May be the comment to add to the property or an array of comments matching the array of property names.
If invalid, the existing comment will be used.

CommitComment

Optional. The comment to be added when the change is committed to the repository.

Deploy

Optional Boolean. Set TRUE to deploy the change immediately.

CallerHasLock

Optional Boolean. Set TRUE if the caller holds the layer lock in write mode.

ClearSection

Optional Boolean. Set TRUE to empty the section (or sections if an array was provided), before writing the new value(s).

Workstation

Optional text. If an integer 0 or 2 was used for the SettingsFile parameter, then the name of the workstation should be provided.
Defaults to the current workstation (WkStaInfo(0)) if missing or invalid.

Comments: This function returns an object which becomes invalid when the operation (handled asynchronously) is complete.

Examples:

Redirect

Description: Redirects a local device to network resource.

Returns: Nothing

Usage: Script Only.

Function Groups: Printer, Software and Hardware, Network

Related to: DefaultPrinter

Format: Redirect(Local, Remote)

Parameters:

Local

Required. Any text expression giving the local device to redirect, for example "G:", "LPT1:", "DEF:" (default printer), etc.

Remote

Required. Any text expression giving the network resource to map the local device to. If this parameter is an empty string or invalid, the current connection for the device Local is disconnected.
If Local has any value other than "DEF:", this parameter must be the same form used by the Windows™ com-

mand prompt, "net use" function:
"\\MyServer\MyPrinter"
If Local has a value of "DEF:", the value of this parameter has three elements separated by commas, as follows:
<printer name>, <driver name>, <port>
Each of these elements must be valid in order to have a valid Remote parameter. If any are invalid, programs such as Print Manager may revert back to the previous valid printer, while other programs may have unpredictable behavior.

Comments: Caution should be exercised when using this statement, since the redirection of the device is permanent – execution of the statement causes the change to be written to the Windows™ registry. For this reason, when using "DEF:" (the Windows™ default printer), it is always a good idea to use the DefaultPrinter function prior to doing the Redirect, so that the original default printer may be restored at a later date.

Example:

```
If ZButton(10, 40, 110, 10, "Print", 1);
[
  oldP = DefaultPrinter() { Keep track of original printer };
  Redirect("DEF:" { The Windows default printer },
         "\\Serve1\NPrinter, HPPCL5MS, Ne00:"
        { The printer on the server });
  PrtScrn() { Do a printout of the screen };
]
```

A press of the button causes the default Windows™ printer to be redirected to the printer called NPrinter on the server called Serve1. The rest of the string indicates the driver to be used in this redirection.

Register (Alarm Manager)

Note: Deprecated. Do not use in new code.

(Alarm Manager module)

Description: Inform the Alarm Manager that a module instance may wish to generate alarms in the future.

Returns: Numeric

Usage: Script Only.

Function Groups: Alarm

Related to: Commission

Format: \AlarmManager\Register(AlarmObject[, Enable]);

Parameters:

AlarmObject

Required. The object value of the new alarm to add to the configured alarm list. A variable called "Name" must exist within the scope of AlarmObject. This variable uniquely defines the text name of the alarm.

Enable

Required Logical Expression. Set to true if the alarm should start enabled. Defaults to "Enabled" if invalid or not defined.

Comments: The Register subroutine always returns "0".
Registering an alarm does not trigger an alarm. It informs the Alarm Manager that the AlarmObject module instance may wish to generate alarms in the future.

Register (Modem Manager)

(Modem Manager module)

Description: This module registers a discriminator that accepts incoming calls.

Returns: Numeric

Usage: Script Only.

Function Groups: Alarm

Related to:

Format: \ModemManager\Register(Context, Station, Priority [, MediaMode, Voice, TimeOutVal, Name]);

Parameters:

Context

An object value of the context in which the discriminator module is to be called.

Station

A unique identifier for this discriminator instance.

Priority

The relative priority of this discriminator to others. Priorities are in the range of 0..99 with numerically higher priorities being called first.

MediaMode

specifies the media mode that this discriminator wishes to handle. Typically, this would specify AUDIO to handle incoming audio calls. The default is DATAMODEM which handles normal data calls. Please refer to the Comments section.

Voice

If this discriminator is to handle AUDIO calls, then this parameter specifies the text GUID of the voice to be used by the Text-to-Speech engine.

TimeOutVal

An AUDIO discriminator generally relies on some human-generated response. This parameter sets the number of seconds that the discriminator runs from before giving up. The default value is 20 seconds. Please refer to the Comments section for further information.

Name

Usually Context will, when cast to a text value, identify the name of a variable in \Code whose value is the object context for the discriminator. If this is not the case, then the name should be given here.

Challenge

A string to send in response to incoming calls that don't initially transmit any data.

Comments:

In order to receive incoming calls, you must first call the Modem Manager's Register method, passing the driver's object value, its station number, and a priority (relative to other drivers).

```
\ModemManager\Register(Root, Station, 10 { Pri-
ority ]);
```

You must also provide a discriminator subroutine. This subroutine will be called by the Modem Manager when it offers you an incoming call. The Modem Manager passes as a parameter a BUFFER which contains the initial data received from the line (see HelloPacketLength). You should parse this data and decide whether or not it is supported by your driver, and for which station it is intended.
Return Invalid to reject this call.
Return the (valid) station identifier to accept the call.
If you accept the call, then the Port\IsConnected() module will go true, and you may acquire the serial port semaphore Port\Sem() and read and write data via the serial port.
If Port\IsConnected() goes false, the call has been disconnected. If you wish to hang up the call, call the subroutine Port\CallComplete().

Register (RPC Manager)

(RPC Manager Library)

Description This subroutine registers a service for RPC and returns a pointer to the variable containing the current RPC status of the service.

Returns Pointer containing the current RPC status.

Usage Script Only. (Subroutine call only)

Function Groups Network

Related to: ConnectToMachine | DisconnectFromMachine | GetServer | GetServersListed | GetStatus | IsClient | IsPotentialServer | IsPrimaryServer | Send | SetRemoteValue

Format: \RPCManager\Register(ServiceName [, ListSource, ListName, SemObj, StartMode, PrioritySync, ServerList, Sticky])

Parameters

ServiceName

Required. The name of the tag to register.

ListSource

The object that is the source of this service's server list. Should have the following callbacks (like LayerModule):

- ServerListSubscribe
- ServerListUnsubscribe
- GetServerList
- GetRPCServiceSettings

Defaults to Layer object with this service's LocalGUID.

ListName

Name of the server list to use for this service.

If invalid, the section name will be ServiceName.
If ListSource is a Layer, then the list used follows the following rules of precedence: service and workstation-specific list, service-specific list, workstation-specific list, then default server list.
If no lists can be found at all, this machine is assumed to be the sole server.

SemObj

An object value for an instance of the system Semaphore module. If Invalid, the RPC Manager will launch an instance of Semaphore for this service.

StartMode

The mode that the service instance will start up in or become when it loses its connection to the server instance. The mode of a service determines whether or not the RPC Manager transmits service broadcast messages from the server.
You should supply the constant \RPC_ACCEPT_ALL (default) if your service requires no synchronization whatsoever. Otherwise, set it to \RPC_SYNC_MODE.
Set TRUE to prevent server from sending Broadcast messages to us.

PrioritySync

When set to non-zero, the PrioritySync flag prevents higher priority servers from syncing with lower priority servers, forcing the lower server to synchronize with itself. Most system services do not set this flag.

ServerList

An optional parameter that is an array of server names for this service. If used, it overrides any server list con-

figuration that has been done in the .RPC files. If not used, the usual method of determining the server list is used.

Sticky

An optional flag that causes this service to stick to a server, even if a higher order server comes online (see also sticky status).

Comments

Subroutine call only.
This subroutine is a member of the RPC Manager's Library, and must therefore be prefaced by \RPCManager\, as shown in the "Format" section. If the application you are developing is a script application, the subroutine call must be prefaced by System\RPCManager\, and the System variable must be declared in AppRoot.src.

The return value from this call is a pointer to a variable that contains the current RPC status of this service instance. Dereferencing this pointer will yield one of the following values
0 – the status of the service instance is unknown at this time.
1 – this service instance is a client to another service instance.
2 – this service instance is the server instance.
Invalid – the caller of Register() has been slain.

This value will almost certainly be required by the service instance's code to allow it to operate correctly.

The ServiceLayer parameter is almost always used for system-level services (services that don't run within running applications). Unless writing a system-level service, leave this parameter as invalid and set the server list via the Edit Server Lists panel of the application properties GUI.

Related Information:

Refer to "RPC API Reference" in the VTScada Programmer's Guide for a listing of Service Control Methods, RPC Methods, and Deprecated RPC Methods.

RegisterCustomTable

(VTSSQLInterface library)

Description: A launched module that registers a name for a virtual database table and defines what information will be available from that table.

Returns: Varies. 0 if array size is invalid,

Usage: Script Only.

Function Groups: Database and Data Source, ODBC

Related to: SQLQuery

Format: \VTSSQLInterface\RegisterCustomTable(Name, Description, Columns, IsReadOnly, CallbackObj, SupportsTPP)

Parameters:

Name

Required. Any text value for the name of the table.

Description

Required. Any text value for the description of the table

Columns

Required. May be either a dictionary or a callback module.

If the first, then Columns will be a dictionary of structures that describing the column contained in the table. The dictionary keys should be the names of the columns. Use this option when the table will refer to a single tag or type of information where the column format is known.

If the second, then a callback context may be provided where the module SQLQueryGetColumnInfo is called to retrieve the column info. If the current module contains this submodule (as is usually the case) then you will use Self() for this parameter.
Use this option if the custom table may refer to multiple tag types, each having its own structure.
Further information is provided in the comments section.

IsReadOnly

Required numeric. Set to 1 for read only. Writes (value 0) are not currently supported through the SQL interface.

CallbackObj

Required. The object value of a module which must contain a submodule named SQLQueryRetrieveData. In most cases, this will be stored in the current file, in which case self() should be used for this paramter.
SQLQueryRetrieveData will be called by VTSSQLInterface\SQLQuery when it needs to retrieve data from the custom table. Further information is provided in the comments section.

SupportsTPP

Required Boolean. Set TRUE if the table has support for additional TPP tables. In this case SQLQueryRetrieveData will be called with a TPP value specified at query-time.

Comments:

Columns Parameter

If the Columns parameter is defined as a dictionary, each element must be a VTSSQLColumnInfo structure. This structure is defined as follows:

```
VTSSQLColumnInfo STRUCT [
                    SQLType { One of the SQL type
constants };
                    IsPrimaryKey { 1 for primary
key column, 0 if not };
];
```

For example, VTScada uses the following code to define the columns when registering the table name, AlarmHistory:

```
AlarmHistoryColumns = Dictionary();
  { Standard fields that will always exist for
the alarm history table }
AlarmHistoryColumns["Timestamp"] = ColumnInfo
(\#SQL_DATETIME, 1, "Timestamp"
AlarmHistoryColumns["Id"]       = ColumnInfo
(\#SQL_VARCHAR, 0, "Name");
AlarmHistoryColumns["Name"]      = ColumnInfo
(\#SQL_VARCHAR, 0, "Name");
AlarmHistoryColumns["SubName"]  = ColumnInfo
(\#SQL_VARCHAR, 0, "Name");
AlarmHistoryColumns["Event"] = ColumnInfo(\#SQL_
VARCHAR, 0);
```

If using a callback object for the Columns parameter, your code must include a module named SQLQueryGetColumnInfo. This subroutine will be called by the SQLQuery module, and must return a dictionary of VTSSQLColumnInfo structures, keyed by column name, for each column that to be queried in the given custom table.

RawTableName

Required. Any text value for the custom table name without _TPP modifiers.

SplitColumnName

Optional text. If specified, the returned dictionary is only guaranteed to return that column (if it exists), but may contain other columns as well. This allows for performance optimization for custom tables that contain large numbers of columns.

SQLQueryGetColumnInfo will normally start with code to strip off any prefix or delimiter from the table name, then obtain the object value of the tag type:

```
{ code to clean up the name followed by...}
TagModuleVal = Scope(\Code, CleanTagTypeName );
```

This will be followed by a call to ListVars to obtain the variables in the specified tag object, then a loop to add each required variable to the dictionary of VTSSQLColumnInfo structures.

CallbackObj Parameter: SQLQueryRetrieveData

SQLQueryRetrieveData is a launched module that must be implemented on the CallbackObj. Note that zero is the proper return if the array size is zero. Its parameters are as follows:

PtrResult

Required. A pointer to the variable, within which the results will be returned.
The returned results must be a 2-dimensional array (even if only one column is returned), indexed in the form: Result[Col][Row]. This variable should not be set until the results are complete.

TableName

Required. The name of the table being queried. This is

required in case the caller decides to register more than one table with the same CallbackObj.

RequestedColumns

Required. An array of the names of the columns needed to satisfy the SQL query.

MaxRecords

Required numeric. The maximum number of result rows that this module is allowed to return.

StartTime

Optional. If the WHERE clause of the query resolves to a starting timestamp, it will be passed here, otherwise Invalid.

EndTime

Optional If the WHERE clause of the query resolves to an ending timestamp, it will be passed here, otherwise Invalid.

TPP

Optional. The time period per row to apply to the query.

FilterHints

Optional. A dictionary containing structured entries for each field in the WHERE clause that can be filtered to one or more ranges or exact values.

The structure definition follows:

```
HintStruct STRUCT [
  Ranges; { Array of RangeDescriptors, field
value must fall into one or more of the
ranges }
  ExactMatches; { Array of values that field
value must match one of. Only valid if all
Ranges have min=max }
  OverallMin; { Overall minimum for the field
value, may be Invalid }
  OverallMax; { Overall maximum for the field
value, may be Invalid }
```

```
];
```

In turn, a RangeDescriptor is a simple structure to contain a min/max range. No distinction is made between inclusive and exclusive ranges. One of min or max may be Invalid, indicating an open-ended range, but not both. For queries that are exact matches, Min and Max will be identical.

```
RangeDescriptor STRUCT [
  Minimum;
  Maximum;
];
```

SQLQueryRetrieveData may ignore any or all of the information received in FilterHints. A simple implementation might use only OverallMin and OverallMax, or just the ExactMatches array. More advanced implementations could use the complete information from the Ranges. Note that ExactMatches, OverallMin, and OverallMax are all derivable from Ranges. They are derived by ParseWhereClause itself for convenience of the consumer.

SQLQuery itself will always apply a post-filter derived from the WHERE expression to filter out any rows that do not match the expression. So SQLQueryRetrieveData need only filter out some of the rows suggested by FilterHints. Filtering out not enough rows will not result in incorrect results returned from the query. (This is also why no distinction is made between inclusive and exclusive ranges in FilterHints. If the range was meant to be exclusive in the original WHERE clause it will be post-filtered correctly.)

Example: for the WHERE clause:

```
... WHERE WellKey="MyWell1" AND Timestamp >=
'2015-06-01' AND Timestamp < '2015-07-01'
```

the dictionary passed to SQLQueryRetrieveData will contain two keys: WellKey and Timestamp. WellKey's value is a HintStruct containing the following elements:

- Ranges – In this example the Ranges array would contain one element of the Range structure with Minimum="MyWell1" and Maximum="MyWell1".
- ExactMatches – In this example ExactMatches array would contain one element set to "MyWell1".
- OverallMin – In this example would be set to "MyWell1".
- OverallMax – In this example would be set to "MyWell1".

Timestamp's value is a HintStruct containing these elements, where the timestamps use the VTSCada timestamp type.

- Ranges – Array(Range('2015-06-01 00:00:00', '2015-07-01 00:00:00'))
- ExactMatches – Invalid
- OverallMin – '2015-06-01 00:00:00' -- Note that if the Timestamp is set to be the primary key on this custom table, this same timestamp value will be passed as StartTime to SQLQueryRetrieveData.
- OverallMax – '2015-07-01 00:00:00' -- Note that if the Timestamp is set to be the primary key on this custom table, this same timestamp value will be passed as EndTime to SQLQueryRetrieveData.

Related Information:

See: "VTScada SQLInterface Module" in the VTScada Programmer's Guide

ReleaseLock

Description:	Releases a working copy semaphore that was acquired by AcquireLock.
Returns:	Nothing
Usage:	Script Only.
Function Groups:	Configuration Management
Related to:	AcquireLock |
Format:	Layer\ReleaseLock()
Parameters:	none
Comments:	Call only you currently have the working copy semaphore.

Examples:

none

RemoveParameter

Description:	Removes a parameter from a module's parameter list.
Warning:	This statement should be used by advanced users only since irrevocable alteration of your application may occur.
Returns:	Nothing
Usage:	Script Only.
Function Groups:	Compilation and On-Line Modifications, Advanced Module
Related to:	
Format:	RemoveParameter(Module, ParmNum)
Parameters:	

Module

Required. Any expression for the module or object value.

ParmNum

Required. Any numeric expression for the parameter to remove, beginning at 1.

Comments: The parameter variable removed by this function becomes a normal local variable.

RemWSDL

(System Library)

Description Disconnects a Realm from a WSDL file and the associated set of VTScada modules, cleaning up any resourced used by that web service.

Returns Nothing

Usage Script Only.

Function Groups XML

Related to: SetWSDL | XMLProcessor | XMLAddSchema | XMLParse | XMLWrite

Format: \System\Webservice\RemWSDL(Realm)

Parameters

Realm

Required. The name of the VTScada Realm to have its web service removed.

Comments Once RemWSDL is called, the associated web service will immediately stop processing messages, however any operations set in motion by that service will run to completion. This function is called implicitly if the connected module is destroyed.

Rename

Description: Renames an existing file.

Returns: Nothing

Usage: Script Only.

Function Groups: File I/O

Related to: GetFileAttribs | SetFileAttribs

Format: Rename(OldName, NewName)

Parameters:

OldName

Required. Any text expression for the current name of the file to be changed. A known path Known Path Aliases for File-Related Functions may be provided in the form, :{KnownPathAlias}.

NewName

Required. Any text expression for the new name that the file is to be changed to. A known path alias may be provided in the form, :{KnownPathAlias}.

Comments: This statement will rename the file regardless of its attributes and the attributes will not be changed. If the file to be renamed does not exist, a file will not be created.

Example:

```
Rename(Concat(appPath, "TestMod.SRC"),
Concat(appPath, "Test1.SRC"));
```

This statement changes the name of a file from "TestMod.SRC" to "Test1.SRC".

Replace

Description: Performs a case sensitive search and replace operation on a buffer and returns the resulting buffer.

Returns: Buffer

Usage: Script or steady state.

Function Groups: String and Buffer

Related to: Locate

Format: Replace(Buffer, Offset, N, Search, Replace)

Parameters:

Buffer

Required. Any text expression giving the buffer to search. The size of the buffer is limited to 65 500 bytes.

Offset

Required. Any numeric expression giving the buffer offset from 0 to start the search.

N

Required. Any numeric expression giving the number of buffer characters (bytes) to search.

Search

Required. Any text expression or array of text expressions, giving the search string(s). Search must be at least 1 byte long. The search is case sensitive.

Replace

Required. Any text expression giving the replace string.

Comments: This function returns a buffer that is the same as Buffer, except that within the first N bytes following Offset, all occurrences of Search are replaced with Replace.

Since the search and replace strings are delimited by quotation marks, to include a set of quotation marks as part of either, you must use two sets of quotation marks inside of the quotation marks that

delimit the string (see example).

Example:

If a variable exists such that:

```
txt = "abcdefABCDEF";
```

And the following statement is executed:

```
txt = Replace(txt { Buffer },
              0 { Start at beginning of buffer },
              12 { Search through 12 characters },
              "fA" { Search for this string },
              "-wow-" { Replace string with this });
```

The result will be that txt will contain the string "abcde-wow-BCDEF". As a further example of how this function is used, if quotation marks were considered illegal characters in a certain context and therefore needed to removed from a string, the following statement could be used to achieve this:

```
txt = Replace(stringWithQuotes { Search buffer },
              0 { Start at beginning },
              StrLen(stringWithQuotes) { Search entire string },
              """" { Find all quotes },
              "" { Replace with nothing });
```

ReplaceStatement

Description:	Replaces a statement with another statement.
Warning:	This function should be used by advanced users only since irrevocable alteration of your application may occur.
Returns:	Nothing
Usage:	Script Only.
Function Groups:	Compilation and On-Line Modifications, States
Related to:	Compile
Format:	ReplaceStatement(NewS, Location, Size, Adjust)
Parameters:	
	NewS

Required. Any expression for the statement value of the new statement. (This value can be obtained using the Compile function.)

Location

Required. Any expression for the code value that indicates what statement to replace. (This value can be obtained using the Compile function.)

Size

Required. Any numeric expression for the size of the new statement text, in characters.

Adjust

Required. Any logical expression. If true, the text in the file will be moved to allow for the statement that is being replaced. If false, the text will not be moved.

Comments: This statement is disabled in the run time version of VTScada. It will do nothing.

ReportError

(System Library)

Description: Post error information to VTS Trace and optionally, to the display.

Returns: Nothing

Usage: Script Only.

Function Groups: Error Manager

Related to:

Format: \System\ReportError(eMsg, QualifierSet[, UserAndSession, Silent, HelpFile, HelpID, TraceCategory, ApplicationGUID])

Parameters:

eMsg

Required. Any text containing the error message.

QualifierSet

Required. An additional error data dictionary. This dictionary can accept any keys and values that can cast to strings. Each error generates a two-column table of information, built using this dictionary. The keys are displayed in the first column and the values are displayed in the second. Note that VTScada system code typically uses Setup.INI values to generate these strings so that they can be changed to suit different languages, but this is not a requirement.

UserAndSession

Optional. This parameter should be generated by a call to \LayerRoot\GetUserID() from the closest GUI-called module to the error. The purpose is to capture the user responsible for the operation that lead to the error. This call is required since users are linked to sessions that are only available to modules ultimately called by the Display Manager (that is, GUI objects). As a result, the "user and "session" often need to be captured earlier and passed to any modules likely to generate errors.

Silent

Optional Boolean. If TRUE, the error dialog will be suppressed. Defaults to the application property, ReportErrorSilenceAll, or FALSE if not otherwise set.

HelpFile

Optional text. If the dialog is to have a help button, and you are using a custom help file, provide its name here. May be left blank if you are using the VTScada help file.

HelpID

Optional numeric. If the dialog is to have a help button, provide the ID value from the help file's Alias file in order to link to the correct topic.

TraceCategory

Optional. The category to use in the VTScada Trace program.

ApplicationGUID

Optional text. Defaults to the current application's GUID. In the rare case where there can be ambiguity over which application caused the error, a GUID may be supplied for the source application.

Comments: If this function displays the error message to the operator, it will use the newer format, which was introduced in VTS 10.0. This format enables you to display multiple error messages in one screen, each with its own Help ID value.
The error will be recorded in the Error Log file at the VTScada level, as well as being stored for use by the TraceViewer. The log file ("errors.log") records every error that is raised via ReportError.

Examples:

```
IfThen(PickValid(!Bit(VarAttributes(FindVariable(VarName, TagMod, 0,
0)), 8 {Temporary}), FALSE),
{ A non-temporary variable of a disallowed name was present. Produce
an error. }
  ErrorInfo = Dictionary();
  ErrorInfo[\TypeNameLabel] = Cast(TagMod, \#VTypeText);
  ErrorInfo[\VariableNameLabel] = VarName;
  ErrorInfo[\IncompatibleVersionLabel] = Ver;
  ReportError(\BadTagVariableErr, ErrorInfo, Invalid, 0, DlgHelpFile,
12009 { HelpID 12009 });
);
```

Related Information:

| See: "Using the Trace Viewer" in the VTScada Programmer's Guide.

RepoSubscribe

Description: Allows the caller to specify a callback which will be triggered whenever the application's repository changes

Returns: Nothing

Usage: Script Only.

Function Groups: Configuration Management

Related to: WCSubscribe |

Format: RepoSubscribe(Subscriber[, Callback])

Parameters:

Subscriber

Required. Scope of the destination for the published messages.

CallBack

Optional. Name of the module to call in the Subscriber when something is published. If invalid, this will default to "RepoNotify".

Comments: The callback is provided with a dictionary of the changed files.
Note that since these changes may not be reflected in the working copy (and therefore not affect the application) the WCSubscribe function is recommended in most cases.

Examples:

Reset

(VoiceTalk Module)

Description: Immediately stops a speech stream and cancels any buffered speech.

Returns: Nothing

Usage: Script Only.

Function Groups: Speech and Sound

Related to: Configure | GetDevices | GetVoices | ShowLexicon | Speak | VoiceTalk

Format: VoiceTalkStream\Reset

Parameters:

VoiceTalkStream

Required. A speech stream returned from VoiceTalk.

Comments: This function returns the error code resulting from issuing the command to the speech engine, or zero if no error was encountered.
Issuing this command will immediately stop all speech for the stream on which it was issued. Other streams will be unaffected. Note that in the process of stopping the speech, a new speech request is issued, so the count of queued VoiceTalk\Speak requests will momentarily increase by one.

ResetParm

Description: Can reset parameters that become latched.

Returns: Nothing

Usage: Script Only.

Function Groups: Compilation and On-Line Modifications, Advanced Module

Related to: NParm | Parameter | PType

Format: ResetParm(Object, Index)

Parameters:

Object

Required. Any object (the object value of a running module instance).

Index

Required. Any numeric expression giving the number of the parameter of interest, starting from 1.

Comments: This statement is for experienced users, and is not needed for normal operation. When this function executes, it attempts to reset Object's parameter at Index. This allows modules to be written which reset their parameters (like the Save statement). This function is useful for resetting TimeOut and MatchKeys functions.

ResultFormat

(ODBC Manager Library)

Description: Subroutine to convert 2-d array as returned from query in the form, Arr[Field][Rec], to a normalized format of Arr [Rec][Field].
For single record reads, this function returns a single dimension array Arr[N] where N is the number of fields.

Returns: Array

Usage: Script Only.

Function Groups: ODBC

Related to:

Format: \ODBCManager\ResultFormat(DataArray, MakeSingle)

Parameters:

DataArray

Required. An array of fields and records to transpose

MakeSingle

Required. If true (1), and if the array size is 1 this indicates that the a single dimension array should be returned.

Comments: This module is a member of the ODBCManager Library, and must therefore be prefaced by \ODBCManager\, as

shown in "Format" above.

ResyncDoc

Description: Synchronizes the time and date for the document and .RUN files.

Warning: This function should be used by advanced users only since irrevocable alteration of your application may occur.

Returns: Nothing

Usage: Script Only.

Function Groups: Compilation and On-Line Modifications, File I/O, Advanced Module

Related to: CanEditDoc

Format: ResyncDoc(Module [, Unsync])

Parameters:

Module

Required. Any expression for the module or object value.

Unsync

An optional parameter that is any logical expression. If true (non-0), the module's .RUN file(s) are forced to be out of sync with the time stamp on its .SRC file(s). If false (0) the .RUN and .SRC files are synchronized. The default is 0.

Comments: This function will set the date and time for the .RUN file to that of the document file.

Return

Description: Sets the return value for the module in which it is executed.

Returns: Nothing

Usage: Script or steady state.

Function Groups: Basic Module

Related to: GetReturnValue | Launch | Slay

Format: Return(X)

Parameters:

X

Optional. Any variable, expression, constant or object value. It can be of any type.

Comments: If a Return statement appears anywhere in the code of a launched module (i.e. one that is executed inside of a script or using the Launch statement), even if that portion of the code is not executed, the module will be considered to be a sub-routine, and will block execution of all other modules in the same thread while it is executing. Execution of the Return statement in a sub-routine results in the module being slain without having to use the Slay statement.

If the Return statement is used in a called module (i.e. one that appears as a statement in a state), execution of the Return statement will not stop the execution of the module.
The return value for the module is set equal to X, and is the same type as the expected value of the module call. The type of the return value is set every time Return is executed, which allows modules to return different types of values during execution if they are called (rather than launched).
This statement may appear both inside or outside of a script, and like other statements that may be

called from a state, if multiple calls are active simultaneously in a module, the return value will be invalid.
Return(); is equivalent to Return(Invalid);

Example:

```
Main [ { State in Module Graphics }
  ZText(500, 100, test , 15, 0);
]
< { Sub-module of Module Graphics }
Test
State1
[
  Return("example of return function");
]
>
```

This sets the return value of the module Test to "example of return function" which is displayed by the ZText statement in state Main of the Graphics module.

Reverse

Description: Returns its parameter with the byte order reversed.

Returns: Varies

Usage: Script or steady state.

Function Groups: String and Buffer

Related to: BuffOrder

Format: Reverse(Type, Value)

Parameters:

Type

Required. Any numeric expression giving the type of the Value parameter.

Type	Meaning
1	Short
2	Long
3	Not supported
4	Text

Value

Required. Any variable, expression, or constant. It can be of any type except object.

Comments: If Value is invalid, the return value is invalid.

Note: After the bytes are reversed, the result is cast to a W_SHORT. If you want an unsigned short for the result, you must AND() it with 0xFFFF.

Example:

```
oldA = 45 { 0b00000000 00101101 };
oldB = "Hello";
newA = Reverse(1, oldA);
newB = Reverse(4, oldB);
```

The value of newA will be 11 520 which is 0b00101101 00000000. NewB will have a value of "olleH".

RibbonCmd

Description: Provides two variables that the ribbon will set when the user activates a command

Returns: Boolean

Usage: Steady State only.

Function Groups: Window

Related to: RibbonContextUI | RibbonGalleryItems | RibbonPersistState | RibbonSetProperty

Format: RibbonCmd(CommandID, CommandData)

Parameters:

CommandID

Required. A variable holding the unique identifier for the activated command.

CommandData

Required. A variable holding any command-specific data.

Comments: The return value of this function can be monitored in order to initiate script execution when the user activates a ribbon command.

Examples:

```
IF RibbonCmd(CommandID, CommandData);
[
  ...
]
```

RibbonContextUI

Description: Displays a mini-toolbar or a pop-up context menu or both at a specified window coordinate.

Returns: Nothing

Usage: Script Only.

Function Groups: Window

Related to: RibbonCmd | RibbonGalleryItems | RibbonPersistState | RibbonSetProperty

Format: RibbonContextUI(CommandID, X, Y);

Parameters:

CommandID

Required. The unique identifier of the command.

X

Required. Horizontal coordinate within the nearest window at which to display the pop-up.

Y

Required. Vertical coordinate within the nearest window at which to display the pop-up.

Comments: UI == user interface.
A context map is an XML declaration of the user interface controls. The XML can hold multiple context maps and therefore, it is possible to select from multiple, pop-up user interfaces to display.

RibbonGalleryItems

Description: Use to populate a gallery with a collection of items or commands, from which the user may make a selection

Returns:

Usage: Script Only.

Function Groups: Window

Related to: RibbonCmd | RibbonContextUI | RibbonPersistState | RibbonSetProperty

Format: RibbonGalleryItems(CommandID, GalleryItems);

Parameters:

CommandID

The unique command identifier of the gallery.

GalleryItems

A structure containing two members:
Categories - A one-dimensional, ordered array of category labels. Each member of the Items array belongs to a category, identifying its category by the index into this array. The categories are displayed as headings within the

gallery. If there are no categories, this member must be Invalid.

Items - A one-dimensional, ordered array of RibbonItem structures. Each entry in the array represents an item or command in the gallery. These are displayed in the same order (within their category) as they appear in this array. RibbonItem structures are described in the comments section.

Comments: A RibbonItem structure has the following members:

Member	Content
Category	The category number of the item or command. Mandatory. If the item does not belong to a category you must specify Invalid here. Failure to do this can result in undefined behavior, such as missing items if the gallery is added to the quick access toolbar.
Label	The text label to be displayed for the item or command. Optional.
Description	The text description to be displayed for list of most recently used (MRU) entries only. This is displayed as the one-line tooltip for an MRU entry. Optional.
Image	The image to be displayed for the item or command. Any VTS-supported image can be supplied as a bitmap value. Optional.
CommandID	For command galleries only, the unique command identifier of the command to represent.
CommandType	For command galleries only, the type of command. This controls how the command will be represented in the gallery.

RibbonPersistState

Description: Persists the state or a ribbon to a VTScada variable.

Returns: Variable

Usage: Script Only.

Function Groups: Window

Related to: RibbonCmd | RibbonContextUI | RibbonGalleryItems | RibbonSetProperty

Format: RibbonPersistState()

Parameters: none

Comments: This function is typically called in a module descructor

Examples:

```
RibbonState = RibbonPersistState();
```

RibbonSetProperty

Description: Set a property on a command.
A ribbon control will access those properties of a command that it needs to render itself and respond to user interaction.

Returns: Nothing

Usage: Script Only.

Function Groups: Window

Related to: RibbonCmd | RibbonContextUI | RibbonGalleryItems | RibbonPersistState

Format: RibbonSetProperty(CommandID, Property, Value);

Parameters:

CommandID

Required. The unique identifier of the command.
If set to zero, the property and value are applied to all elements of the ribbon, allowing all to be

disabled or enabled with a single call.

Property

Required. An integer from the table that follows.

Value

Varies according to the property being set.

Property Value	Property Meaning	Value Type
0	Enable/Disable the command. If the command is disabled, all control representations of the command will be rendered grayed and the user will be unable to activate the command.	Boolean: 0 (Disable); 1 (Enable)
1	Boolean value of the command. For example, a command intended to be represented by a toggle button or check-box would have Boolean value.	Boolean
2	Numeric value of the command. Commands represented by a combo-box or spinner control use numeric values.	Integer or double
3	String value. Commands represented by controls that display a string value use this to provide the value.	Text
4	Minimum value. Commands represented by a spinner control use this to regulate the minimum value the user can select.	Integer or double
5	Maximum value. Commands represented by a spinner control use this to regulate the maximum	Integer or double

	value the user can select.	
6	Label. The label displayed by a control that has a textual representation of the command, for example the text next to a button.	Text
7	Description. The description displayed by a control that has a long textual description of a command, for example, a "MajorItems" button in a DropDownButton control.	Text
8	Tooltip Title. The emboldened title displayed in a tooltip.	Text
9	Tooltip Description. The textual description displayed in a tooltip. This is not emboldened and is displayed below the tooltip title (if any).	Text
10	Key tip. Text that is displayed to assist the user when a key combination is bound to the key, e.g. "F9" or "Alt+U".	Text
11	Small Image. The small (?x?) image displayed by a control.	Image (see comments)
12	Large Image. The large (?x?) image displayed by a control.	Image
13	Color. An RGB quad value for a command that is intended to be represented by a color picker control.	Integer
14	Color Type. An enumerated type for use by a command that is intended to be represented by a color picker control. Normally	Integer as follows: 0 == no color

	used in conjunction with the Color property.	1 == automatic color 2 == RGB color value
15	Selected Item. Used for item galleries to force a specific item to be drawn as selected.	Integer
16	Context Available. Commands whose representation is a contextual tab group have their visibility controlled by this.	Boolean
17	Font. Commands represented by a font selection control have the values for the fields in the font control set by this property.	A font command structure
18	Representative String. Supply a string for combo-box or spinner controls to set their width. The string will be used to measure how wide the control will be.	Text
19	Increment. For spinner controls, the increment or decrement applied when the up or down arrow buttons are clicked.	Integer or double
20	Decimal places. For spinner controls, the number of decimal places displayed.	Integer

Comments: Note that setting a property on a command affects all controls that use the changed property of the command.
Images displayed on buttons and other controls that accept images can be defined by the XML markup or via script code. Images supplied via markup are restricted to Windows .BMP 32-bit ARGB format for supported Windows systems up to and including Windows 7. From Windows 8, .PNG are also sup-

ported.
Images supplied via script code can be of any image type supported by VTScada and are internally converted to the appropriate type for the ribbon.
Image size can be hard to define. The exact image size expected by the ribbon framework depends on the resolution the display device is set to. There are two image sizes used by the ribbon, denoted as Large and Small, as follows:

Display resolution (DPI)	Small Image size (pixels)	Large Image size (pixels)
96 dpi	16x16 pixels	32x32 pixels
120 dpi	20x20 pixels	40x40 pixels
144 dpi	24x24 pixels	48x48 pixels
192 dpi	32x32 pixels	64x64 pixels

RmDir

Description: Destroys a directory on a disk and returns its own error code.

Returns: Numeric

Usage: Script Only.

Function Groups: File I/O

Related to: MkDir

Format: RmDir(Name [, DelAll])

Parameters:

Name

Required. Any text expression that is the full path name of the directory to delete. A known path Known Path Aliases for File-Related Functions may be provided in the form, :{KnownPathAlias}.

DelAll

An optional parameter which, when set to 1, causes all files and subdirectories of the named directory to be deleted. Default: 0

Comments: The return value is 0 if successful and -1 otherwise.

Note: If DelAll is not set to 1 and if the directory contains files or subdirectories, then the directory will not be deleted.

If Name is given as a relative path, then VTScada will look for that directory starting in whatever directory holds the module that is making the RmDir call. Thus RmDir("Sample"), run from within an application that is located in the directory C:\VTScada\MyApp, will remove the directory C:\VTScada\MyApp\Sample.
Note that this behavior differs from that of the MkDir function.

Example:

```
err = RmDir("C:\Sample");
```

If possible (permissions permitting), directory Sample on the C drive will be deleted and err will be set to 0. If unsuccessful, err will be -1.

RootTransform

Description: Returns the object value that contains the root transform applied to the given module.

Returns: Object

Usage: Script or steady state.

Function Groups: Advanced Module, Graphics

Related to: GUITransform | UnTransform

Format: RootTransform(Object)

Parameters:

Object

Required. Any expression which returns an object value. This is the object value for which the root transform is being sought.

Comments: The return value may be the same as Object if Object contains the transform.

Example:

```
transMod = RootTransform(Self());
```

RootValue

Description: Retrieves the root value from a dictionary. This function will always attempt to return a value that is not itself a dictionary. If the value stored as the root of the given dictionary is also a dictionary, this function will return the root value from that second dictionary. Should all root values be other dictionaries (which would imply that the dictionary at the end of the chain must actually be an earlier dictionary) then RootValue will traverse the chain until it finds a root value which is an earlier dictionary (i.e. the end of the chain before it loops back) and will return that root value. This is the only situation where the command will return a dictionary as the result.

Returns: Varies

Usage: Script or steady state.

Function Groups: Dictionary, Variable

Related to:

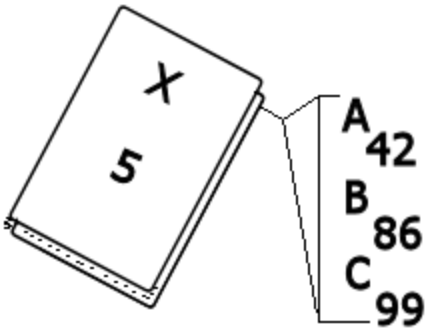
Format: RootValue(dictionary);

Parameters: Dictionary | MetaData | DictionaryCopy | DictionaryRemove | GetNextKey | GetKeyCount | HasMetaData | IsDictionary | ListKeys

Dictionary

Required. The name of the dictionary to find the root value of.

Example:



(where the dictionary illustrated is named "X")

```
RVAL = RootValue( X );
RVAL == 5;
```

Note: You can always access the root value of a dictionary directly using the syntax Y = X[""]. This technique will not follow links to other dictionaries and will always return the root value of the requested dictionary, even if that value is another dictionary.

RootWindow

Description: Returns the object value of the root (original) module displayed in the same window.

Returns: Object

Usage: Script or steady state.

Function Groups: Basic Module, Window

Related to: ParentWindow | Window

Format: RootWindow(Object)

Parameters:

Object

Required. Any expression that returns an object value. This is the object value where the search starts for the root window.

Comments: The return value may be the same as Object if Object is the root module of its window.
For modules in non-child windows (i.e. one without bit 9 set), RootWindow and ParentWindow will return the same value. For child windows, RootWindow will return the root module in the child window, while ParentWindow will return the root module in the child window's closest non-child calling window.

Example:

```
rootWin = RootWindow(Self());
```

Rotate

Description: Returns a Rotate value, which specifies a rotation about a point.

Returns: Rotate

Usage: Steady State only.

Function Groups: Graphics

Related to: Normalize | Point | Trajectory | Vertex

Format: Rotate(Amount, MinDegrees, MaxDegrees, Center)

Parameters:

Amount

Required. Any expression that returns a Normalize value, specifying how much to rotate.

MinDegrees

Required. Any numeric expression giving the minimum amount of rotation, in degrees. This is not a limit on the amount of rotation.

MaxDegrees

Required. Any numeric expression giving the maximum amount of rotation, in degrees. This is not a limit on the amount of rotation.

Center

Required. Any expression that returns a Point. This is the center point for the rotation.

Comments: The return value is a Rotate value which specifies how to rotate about a point. It may be used in any function that accepts a Rotate value, and specifies how to rotate that object.

Example:

```
armRot = Rotate(Normalize(armEncoderPosition, 0, 100)
                { Amount of rotation },
                45 { Minimum rotation, in degrees },
                135 { Maximum rotation, in degrees },
                Point(50, 75, Invalid, Invalid)
                { Center point for rotation });
```

This specifies a rotation about the point (50, 75). If armEncoderPosition is 0, any object that uses this Rotate will rotate 45 degrees counter-clockwise about the point (50, 75). If armEncoderPosition is 100, any object that uses this Rotate will rotate 135 degrees counter-clockwise about the point (50, 75). If armEncoderPosition is any other value, the rotation will be by a proportional amount, not limited to the range 45 to 135 degrees.

RTimeOut

Description: Cumulative Timer. This function returns true when the total time that an expression is true reaches the specified value.

Returns: Boolean

Usage: Steady State only. See: Rules for Usage.

Function Groups: Time and Date

Related to: AbsTime | Latch | TimeOut | Toggle | Save

Format: RTimeOut(Enable, Time)

Parameters:

Enable

Required. Any numeric expression giving the condition that results in the timer counting. When this parameter is true (not 0), the timer is "running." When this parameter is false (0), the timer stops but the total time accumulated so far is maintained as the point which counting will start when the parameter becomes true again.

Time

Required. Any numeric expression giving the time-out limit in seconds. When the cumulative time that Enable is true reaches this value, the function becomes true (1).

Comments: This function is reset when either parameter becomes invalid or when the state containing the function is started. When the function is reset, counting starts at 0 and the returned value is false (0). Note that this function is reset automatically when it occurs in a true action trigger or function parameter of a function which resets its parameters after evaluation (e.g. Latch, Toggle or Save).

Example:

```
ZText(10, 10 { X-Y coordinates },
      Cond(RTimeOut(motorRunning, 28800),
                    "Check motor - 8 hrs running time accumulated",
                    ""),
      0, 0 { Black text in default font });
```

This statement displays a message when the variable motorRunning has been true (any non-zero value) for 28800 seconds (8 hours).

Related Information:

See: "Latching and Resetting Functions" in the VTScada Programmer's Guide

RUNFileName

Description: Returns the name of the .RUN file for a module including the full drive and path.

Returns: Text

Usage: Script or steady state.

Function Groups: Compilation and On-Line Modifications, File I/O, Advanced Module

Related to: ChildDocs | Dir | FileFind | ModuleFileName

Format: RUNFileName(Module)

Parameters:

Module

Required. Any expression for the module or object value.

Example:

```
{ Find all modules including root; recurse in }
allMods = ChildDocs(Self(), 1 + 2 + 8);
numMods = ArraySize(allMods, 0);
{ Find the .RUN file names for all modules }
RUNList = New(numMods);
i = 0;
WhileLoop(i < numMods,
         RUNList[i] = ToUpper(RUNFileName(allMods[i]));
        i++;
);
```

This group of statements will result in the file names for all .RUN files for the current application being stored in the array called RUNList.

RUNFileVersion

Description: Returns the minimum version of VTScada that can read the .RUN files produced by the current version.

Returns: Text

Usage: Script or steady state.

Function Groups: Compilation and On-Line Modifications, File I/O, Advanced Module

Related to: Version

Format: RUNFileVersion( )

Parameters: None

Example:

```
ZText(10, 10, Concat("Minimum version to read .RUN files is ",
                     RUNFileVersion()), 0, 0);
```

This statement will display the .RUN file version in the upper right corner of the window along with a comment about what the number represents.

RunPack

(RPC Manager Library)

Description: Unpacks and executes a set of RPCs from a stream constructed with PackRPC.

Returns: Text

Usage: Script Only. (Subroutine call only)

Function Groups: Advanced Module, Network, Stream and Socket

Related to: PackRPC

Format: \RPCManager\RunPack(Stream [, Service]);

Parameters:

Stream

Required. A packed RPC stream obtained from one or more PackRPC method calls.

Service

The name of the service to be used to determine root

scope for the RPCs within Stream. Invalid for non-service RPCs. If valid, the RPC subroutines specified in the package will be searched for starting in the scope of the service instance of the machine on which the RunPack() RPC is executing.

Comments: This subroutine is a member of the RPC Manager's Library, and must therefore be prefaced by \RPCManager\, as shown in the "Format" section. If the application you are developing is a script application, the subroutine call must be prefaced by System\RPCManager\, and the System variable must be declared in AppRoot.src.

S Functions

The sections that follow identify all VTScada functions beginning with "S".

Save

Note: Deprecated. Do not use in new code.

Description: This threaded function stores data in a circular historical data file at times indicated by a condition and returns the record number last written to disk.

Returns: Numeric

Usage: Steady State only.

Function Groups: File I/O

Threaded: Yes

Related to: Save | HistorianDeleteRecords | HistorianGetData | HistorianGetInfo | HistorianReadRecords | HistorianWriteRecords | Get | GetHistory | GetLogInfo | SaveHistory | TGet

Format: Save(NStatus, NByte, NShort, NLong, NFloat, NText, TSize, Records, Buffers, File, Trigger, V1, V2, ...)

Parameters:

NStatus

Required. Any numeric expression giving the number of status type values to store in the file. Any functions used in this parameter must be able to be executed in a script, since triggering of the Save re-evaluates it as if it were in a script.

Status types must be equal to a 0 or a 1. This value must be greater than or equal to 0.

NByte

Required. Any numeric expression giving the number of values that are only one byte long. Any functions used in this parameter must be able to be executed in a script, since triggering of the Save re-evaluates it as if it were in a script.

Byte types must be in the range 0 to 255 inclusive. This is a subset of the short type but require 50% less file space than short values and 75% less space than either long or float values. This value must be greater than or equal to 0.

NShort

Required. Any numeric expression giving the number of short type values to store in the file. Any functions used in this parameter must be able to be executed in a script, since triggering of the Save re-evaluates it as if it were in a script.

This value must be greater than or equal to 0.

NLong

Required. Any numeric expression giving the number of long type values to store in the file. Any functions used in this parameter must be able to be executed in a script, since triggering of the Save re-evaluates it as if it were in a script.

This value must be greater than or equal to 0.

NFloat

Required. Any numeric expression giving the number of float type values to store in the file. Any functions used in this parameter must be able to be executed in a script, since triggering of the Save re-evaluates it as if it were in a script.

This value must be greater than or equal to 0.

NText

Required. Any numeric expression giving the number of text type values to store in the file. Any functions used in this parameter must be able to be executed in a script, since triggering of the Save re-evaluates it as if it were in a script.

This value must be greater than or equal to 0.

TSize

Required. A short constant giving the number of characters to reserve for text values. Text values that are longer than this number will lose the characters that fall beyond this limit. All text values stored will have the same size.

Records

Required. Any numeric expression giving the number of entries in the file. If the file is to be created full size (to prevent fragmenting of the file as it grows larger), the size should be multiplied by -1.

Since the file is circular in nature, once this number of records have been written, new records will overwrite old ones, beginning with the oldest record in the file.

Buffers

Required. Any numeric expression for the number of records to keep in memory (RAM) before writing to

disk. A value of 0 will result in the data being directly written to disk each time Trigger is true. Setting the buffer to large numbers speeds the average data logging rate substantially but requires RAM equal to the number of records specified by Buffers.

When the RAM buffer is full the entire buffer is written to disk. If the buffer size is changed or if the Save is stopped (such as when VTScada stops), any data in the buffer is immediately written to disk.

Also, if a Get is executed, any data in the buffer will be immediately written to disk, so that the user in ensured of getting all current data.

Normally when Save stops, a record with all invalid fields, and the current time and date stamp, is written to the file. This indicates that data are no longer logged, and fields are unknown until Save resumes. If Buffers is negative, this invalid record is not written, and the number of buffered records is -Buffers - 1. This is to remain compatible with prior versions of VTScada.

Be aware that any data held in a RAM buffer will be lost if VTScada is not correctly stopped (for example, a power failure).

File

Required. Any text expression giving the file name for the historical data. Any path name, including any special, remote, and networked drive is allowed. The default extension is ".DAT" and the default path is the current application path (where VTScada was started). If the file name is prefixed with a period, the path will be to the directory the module is contained in.

Trigger

Required. Any resettable function whose transition from false to true indicates the data items V1, V2, ...

are to be written to the file. The Save statement resets the Trigger parameter if the Trigger is true (not 0).

V1, V2, ...

Required. A series of expressions which give the values to be stored in the file. The parameters must be in the order: status, byte, short, long, float, and text. The number of each type is given by the NStatus, NByte, NShort, NLong, NFloat and NText parameters.
There must be exactly the number of Vn parameters specified by each of these parameters.

Comments: This function returns the record number last written to disk. Since this function is threaded and runs as a background job, this value can be used to determine when data have actually been written by examining the change in the return value.

This function creates the historical data file when it is entered during configuration. If the file already exists and is of a different format than specified by the statement, the program automatically converts the file to the new format. This conversion will result in data loss if the number of values of any type is reduced. For example, if NStatus is reduced from 16 to 12, the last four status parameters stored in the file before the change will be lost. If a value type number is increased or remains the same, no data will be lost.

When a Save function is triggered, the next buffer is filled. When the number of filled buffers equals the value implied by the parameter Buffers the file is opened, written, and then closed. This allows a file to be logged to a network server, and read by other VTScada applications on the network. The VTScada NetBIOS I/O driver is not required for this type of network communication. All that is required is a network redirector.

If a file is referenced by two or more active Save functions, there will be unpredictable results, with a possibly corrupted file and reduced performance. Do not reference the same file from more than one Save function at the same time. This is not the case with a SaveHistory, Get or TGet statement. Any calls to SaveHistory will result in data being writ-

The values of the Vn parameters need not be valid for the Save statement to operate. Any invalid values are recorded as such in the file. The data are stored in the file in the order given by the Vn parameters. This order is important for retrieving the data using the Get statement.
Note that the Trigger parameter is reset when it becomes true. This means that data can be saved at regular time intervals by using TimeOut or AbsTime as the trigger. Once TimeOut becomes true, the timer will be restarted by the Save statement. Change can also be used in Trigger to cause data to be stored when data values change by a given amount. The starting point for the Change function will be reset once the Trigger becomes true and the data are stored.
The size of the file can be calculated as follows:

```
Record *
Ceil(Ceil((NStatus + NByte + NShort + NLong + NFloat + NTex-
t)/8) +
Ceil(NStatus / 8) + NByte + 2 * NShort + 4 * NLong + 4 *
Nfloat +
(TSize + 2) * NText + 8)
+ 25
```

For example, if the Save statement was as follows :

```
Save(3, 1, 4, 0, 6, 1, 24, 10000, ...
```

The file size in bytes would be :

```
10000 * (Ceil((3 + 1 + 4 + 0 + 6 + 1) / 8) + Ceil(3 / 8) + 1
+
2 * 4 + 4 * 0 + 4 * 6 + (24 + 2) * 1 + 8)
+ 25
= 10000 * (Ceil(15/8) + Ceil(0.375) + 1 + 8 + 0 + 24 + 26 +
8)+ 25
= 10000 * (2 + 1 + 67) + 25
= 700,025 bytes
```

The size of the files generated by Save must be considered so the available disk space is not exceeded. Note that the files grow as data is recorded up to a maximum of the calculated size. To prevent fragmenting of the file as it grows over time, it may desirable to create the file full size, by making the Records parameter negative.
It is good practice to create several log files of varying frequency. For example, you may want to log data to a file every

Another way to decrease logging frequency is using a Trigger based on Change, if the values logged to the file change significantly only rarely.

File Format:The standard file format for Save files is a 38 byte header, followed by identical format records (as specified by the parameters). The header contains the following information:

Bytes	Size	Description
0-1	Int	Number of status values
2-3	Int	Number of bytes values
4-5	Int	Number of short values
6-7	Int	Number of long values
8-9	Int	Number of float values
10-11	Int	Number of text values
12-13	Int	Text size
14-17	Long	Maximum number of records in file
18-	Long	Index of next record to write

21			
22	Byte	Bits	Meaning
		0	File has not wrapped around
		1	File has wrapped around
		2	File has not wrapped but was created full size
		3-5	File format version
		6	Checked flag. This bit is set when the file has been checked for inconsistent timestamps. If this file is opened by the Save, SaveHistory, or the ValidateHistory function, the invalid record is updated, and the bit is set.
		7	Clean flag. This bit is cleared whenever data is written to the file by the Save, SaveHistory, or ValidateHistory function. When the Save function is stopped, the invalid record is updated, and the bit is set. If the file is opened by the Save, SaveHistory, or ValidateHistory function, the bit is checked, and if not set, the invalid record at the end of the file is corrected
2-3	Int	Length of a record (in bytes)	
24			
25		Reserved	
2-6-	Long	Actual size of header	
29			
30-	Double	A timestamp for determining the latest time for which valid data was written.	
37			

The first 8 items in the header are the values in the parameters of the Save statement that created the file. See the Save statement parameters for a more detailed description.
Bytes 18-21 are a long number that indicates where the next record will be written in this file. Record numbers begin at 0. To find the file offset (in bytes) where the next record will be written, multiply this number by the length of a record (in bytes), and add 25. Thus record 0 will be found at offset 25.
Byte 22 is a flag. If 0, the circular file has not been filled (the size of the file continues to grow). If 1, data have been overwritten and the last record index has "wrapped-around" to position 0 at least once.
Bytes 23 to 24 are an integer that specifies the length of a record in bytes.
Note that the size of the file header may be increased by using the SetLogHeader function, which will cause additional bytes to be added to the end of the header. Information may then be written to the header using an FWrite (or similar) function with a starting offset of 25. If the header size is to be expanded and written to, however, it is crucial that the Save statement not be active at the time, and also that the data written to the expanded header not exceed the size by which the header was increased. If either of these conditions is not met, the file will become corrupt and data may be lost.
25 Reserved
26-29 Long actual size of header in this file
30-37 Double a special timestamp value which, in the event that the file is not closed tidily, allows the latest time for valid data to be determined.
In addition, bits 3-5 of byte 22 have the value 1 to indicate the file format version.
Note that the size of the file header may be increased by using

the SetLogHeader function, which will cause additional bytes to be added to the end of the header. Information may then be written to the header using an FWrite (or similar) function with a starting offset of 38. If the header size is to be expanded and written to, however, it is crucial that the Save statement not be active at the time, and also that the data written to the expanded header not exceed the size by which the header was increased. If either of these conditions is not met, the file will become corrupt and data may be lost.

Record Format:Each record consists of the following parts:

1. The first 8 bytes of each record are a double precision number in IEEE format, which indicates the time and date when the record was written, as the number of seconds since midnight on 1 January, 1970.
2. Next, 1 byte is allocated for each 8 data items in the record (or fraction thereof). The bits in each byte indicate whether each item is valid (1) or invalid (0).
3. Data items for the record are written sequentially. Each data type is written in a particular Format:1 byte is allocated for each 8 status items or fraction thereof. Each bit contains 1 status value.

Byte items are written as unsigned integers (1 byte each)

Short items are written as signed integers, low byte first (2 bytes each).

Long items are written as signed integers, low byte first (4 bytes each).

Float items are written in single precision IEEE floating point format (4 bytes each).

Text items consist of 2 + TextSize bytes each. The first 2 bytes indicate the number of meaningful data bytes to follow. All bytes beyond that number contain random data. The data bytes are not null-terminated. If, however, TextSize has the value 0, then this indicates that variable length text storage is required.

In this case each text entry in the file has a fixed size of 10 bytes and the actual text is stored in a separate file which has the same root filename as the .DAT file, but has a .STR extension.

It should be noted that the Save statement accepts calculated values for its first six parameters. This allows a more efficient generic data logger to be written. The parameters to be logged are treated as if they were in a script so that the changes to these values do not retrigger the Save (thereby saving time and RAM, since this retriggering is always redundant). The Save statement treats assignments in the first six parameters (which is very rare), so that they will only be executed when the Save trigger becomes true. Only expressions that are allowed in scripts are permissible for the first six parameters.

Example:

```
Save(1 { Number of status values },
     0, 0, 0 { Number of byte, short, long values },
     numFloat { Number of floating point values },
     1 { Number of text values },
     32 { Size of text value },
     1152 { Number of records (4 days worth) },
     50 { Buffer 50 records before writing },
     "G:\DATA\REACTOR" { Path and file name (default .DAT) },
     AbsTime(1, 300, 0) { Save every 5 minutes },
     valveOpen { Status value },
     mixerTemp { First float value },
     inletTemp { Second float value },
     outletTemp { Third float value },
     coreTemp { Fourth float value },
     chemName { Text value });
```

This stores 1 status, NumFloat float and 1 text value to a file on G: (a shared network drive) in directory DATA called REACTOR.DAT. Only the first 32 characters of the text value chemName are logged. Values are logged every 5 minutes, and written to disk every 250 minutes (50 records). The number of records has been chosen based on the fact that we want to have exactly 4 days worth of data. This means that we need 4

* 24 * 60 / 5 = 1152 records in the file. Since it may also be important to know how large this file is, we can calculate the size as follows:

```
1152 records *
[ Ceil((1 status value + 4 float values + 1 text value) / 8) +
  Ceil(1 status value / 8) + 4 * 4 float values +
  (32 chars + 2) * 1 text value + 8
]
+ 25
= 1152 * (Ceil(6/8) + Ceil(1/8) + 16 + 34 + 8) + 25
= 1152 * (1 + 1 + 16 + 34 + 8) + 25
= 1152 * (60) + 25
= 69 145 bytes
```

Related Information:

See: "Latching and Resetting Functions" in the VTScada Programmer's Guide

SaveHistory

Note: Deprecated. Do not use in new code.

Description: This threaded function saves an array of data to a .DAT file for a certain time span.

Returns: Nothing

Usage: Script Only.

Function Groups: Array, File I/O, Log

Threaded: Yes

Related to: SaveHistory | HistorianDeleteRecords | HistorianGetData | HistorianGetInfo | HistorianReadRecords | HistorianWriteRecords | Get | GetHistory | GetLogInfo | Save | TGet

Format: SaveHistory(Array2DElem, Num, File, [Mode], [PathPrefix], [EndTIme])

Parameters:

Array2DElem

Required. Any array element giving the starting point into the two dimensional array containing the data to

write to the file. The subscripts for the array may be any numeric expression, but both must be specified. The format of the array is the same as that returned by GetHistory: [column][record].

Column 0 is the timestamp in seconds since January 1, 1970, while each subsequent column is the data for that record in the same order used by the Save function that created the .DAT file.

Extra columns are ignored, but if there are fewer columns than required to specify all fields in the .DAT file then the function fails and does nothing.

Num

Required. Any numeric expression giving the number of records to write to the file.

File

Required. Any text expression giving the file name for the historical data file. If this file does not exist the SaveHistory function will create it with default format (see Comments). The file extension is optional and will default to ".DAT" if omitted.

Mode

Historical data files use an invalid record to mark discontinuities in the recorded data. Under certain circumstances, it may be desirable to manipulate these records. This optional parameter provides the mechanism to do so. The mode parameter has the following options: (Defaults to 0)

Value	Meaning
0	Add data after the previous record according to time stamp, and don't write an invalid record (default);
1	No longer used
2	Add data according to time stamp. If previous record is invalid, overwrite it, write data, and then write an invalid record after it.
3	Perform a tidy closedown of a file which has been written with Mode = 2.
4	Specifies that SaveHistory should perform a special merge of the supplied data into the referenced data file.

The aim of the special merge is remove invalid records unless the existing and the incoming data both agree that they should exist in the file. The aim of this mode is to assist the resolution of data at primary and backup log servers.

PathPrefix

An optional text expression parameter that enables and controls the retrieval of data from across a set of files.

EndTime

Specifies the end time range for a special merge (Mode = 4). EndTime is optional but may only be specified when Mode = 4.

If omitted or if a value of zero is used, then EndTime defaults to the last time stamp in the records supplied in Array2DElem.

Comments:

Each record of data in the array will be inserted into its correct position based on its timestamp. For this reason the transfer to the file will be more efficient if the timestamps are in increasing chronological order, however, any order is allowed. Any records that have an invalid timestamp will be skipped. Since this function is threaded and runs as a background job, there is no way to tell when the data has all been written to the file other than by checking its contents, however, multiple Save or SaveHistory commands will get written to the file in correct locations based on their timestamps. If a Get or TGet statement is executed, the SaveHistory statement will write all of its data to the file prior to the data retrieval. This also occurs if the application is stopped.

If the target file does not exist, SaveHistory will create it. To do this, SaveHistory needs to determine the column structure of the file. Whereas Save has parameters allowing the format to be explicitly defined, there are no equivalent parameters to SaveHistory. Instead, SaveHistory will analyze the first record that is to be written and determine the parameter type required for each column. In order to ensure that the correct format is determined, it is advisable to make the first record a template record. A template record would have an invalid timestamp followed by column values that unambiguously identify the range of values for each column (see the Save function for a description of the data types). In the case of text values, the template string should be the same length as the maximum

required length.

If Mode has the value 2, then the understanding is that a series of SaveHistory functions are to be performed on the file and that each will be written at the end of the file (if the write is not at end of file, then Mode = 0 is forced). The invalid record that is written at the end of the file identifies that the file is still (conceptually) open. Should the system suffer any form of abrupt shutdown, then this status will be preserved in the file and will be of use in learning the validity of file data. The final SaveHistory to such a file should be written with Mode = 3. This will change the invalid record to one which signifies that the file is no longer open.

If Mode has the value 4, then SaveHistory compares the data supplied in Array2dElem with the data already existing in the file, for the time range specified by the earliest record in the data and the value of EndTime.
The comparison reviews the validity of each record (a record is deemed invalid if all columns are invalid) and splits both sets of data into periods of validity and invalidity. The resulting file will contain all valid records from both data sets and those invalid records where both data sets saw an invalid.

SaveImage

Description: Takes an image handle and saves it to an image file on disk.

Returns: Handle to image

Usage: Script Only.

Function Groups: Graphics

Related to: CaptureImage | MakeBitmap | ModifyBitmap

Format: SaveImate(BitmapHandle, Filename, [MIMEType, Width, Height])

Parameters:

BitmapHandle

Required. The handle to the image to be saved.

Filename

Required. Any text expression for the name to be given to the new file.

MIMEType

Optional text for the format to be used. Defaults to "image/png" if not specified.
Options include "image/bmp", "image/jpeg", "image/-gif", "image/tiff", and "image/png".

Width

Optional. Any numeric expression for the width of the image in pixels. Defaults to the native size of the image.

Height

Optional. Any numeric expression for the height of the image in pixels. Defaults to the native size of the image.

Comments:

Examples:

SaveImage can be used in conjunction with ModifyBitmap. For example the following code snippet can be used on a VTScada application page (in script):

```
SaveImage(ModifyBitmap(CaptureImage(Caller(Self())),
            FALSE { Mirror },
            0 { Hue },
            Invalid { Saturation },
            Invalid { Lightness },
            Invalid { Transparency },
            Invalid { Contrast },
            Invalid { ColorizeHue },
```

```
            Invalid { ColorizeIntensity},
            TRUE { AntiAlias }),
            "c:\temp\page.png", "image/png", 1000, 500);
```

To capture an image of the page, modify it to enable anti-aliasing so that high-quality interpolation will be used for the downsizing, and save it as a 1000x500 PNG.

SaveModule

Description: Saves a module definition to its *.RUN file.

Warning: This statement should be used by advanced users only since irrevocable alteration of your application may occur.

Returns: Nothing

Usage: Script Only.

Function Groups: Compilation and On-Line Modifications, Advanced Module

Related to:

Format: SaveModule(Module)

Parameters:

Module

Required. Any expression for the module or object value.

Comments: This statement saves a module definition to its *.RUN file.

Scale

Description: Returns a value that has been converted from one scale to another.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Generic Math

Related to: Normalize | Cond | Limit

Format: Scale(Value, In1, In2, Out1, Out2)

Parameters:

Value

Required. Any numeric expression giving the value to be scaled.

In1

Required. Any numeric expression giving the minimum of Value's unscaled range. This parameter corresponds to Out1. This is usually the "zero" for the unscaled Value.

In2

Required. Any numeric expression giving the maximum of Value's unscaled range. This parameter corresponds to Out2. In2 must not equal In1.

Out1

Required. Any numeric expression giving the minimum of Value's scaled output range. This parameter corresponds to In1. This is usually the "zero" for the scaled Value.

Out2

Required. Any numeric expression giving the maximum of Value's scaled output range. This parameter corresponds to In2.

Comments:

This function may be used in combination with Cond to perform piecewise linearization of a value or expression. Limit may be used to keep the result within bounds.
The compiler will reduce this function to a constant if all of its parameters evaluate to constants.

Examples:

Suppose the top edge of a bar on the screen is to go from 10 to 527 for a corresponding process value change of -50 to +100. The scale function parameters would be :

```
Scale(Value, -50, 100, 10, 527);
```

It should be noted that the Value parameter need not remain within the range of In1 to In2 for the result to be valid. For example, to convert from Fahrenheit to Celsius temperature scales :

```
Celsius = Scale(Fahrenheit, 32, 212, 0, 100);
```

This means that 32 F = 0 C and 212 F = 100 C. If the temperature were outside the range of 32 to 212, the resulting Celsius temperature would still be correct.

Scope

Description: Performs a scope resolution and returns a reference to the requested member within a module or other object.

Returns: Reference

Usage: Script or steady state.

Function Groups: Basic Module

Related to: Variable | ScopeLocal

Format: Scope(Object, Member[, ScopeLocal])

Parameters:

Object

Required. Any expression for the object value (module) where Member may be found.

Member

Required. Any text expression for the member name. This must be a simple variable or module. Array references or further scope resolution are not allowed inside the text expression.

ScopeLocal

Optional. A Boolean expression. Defaults to FALSE if missing, invalid or if a non-Boolean is provided.
If set TRUE then the call will not search up the scope tree for name matches.

Comments: This function is the same as the '\' operator, when the '\' operator is used between two operands. (Object\Member). Unlike the backslash operator, the Scope function allows any text string to be used. For members whose names have been obfuscated, Scope() offers the only means of referencing them.
This function may be used as a value, or as an L-value (on the left hand side of an assignment). This function is used to reference one specific occurrence of a variable in a module, from another module.
If the final result is Invalid, this function looks for the presence of backslash (\) characters in the second parameter and parses the result.
When searching for a match, this function ignores variables with the PROTECTED attribute. That is, if such a variable is encountered the search will simply skip over it and continue. Detecting variables with the PROTECTED attribute requires use of the 'Variable' function

Example:

```
Scope(\Code, "TagName");
```

Returns a reference to the given tag object, found within the current module.

Using ScopeLocal. Start with the following (noting that the initial example could be written more efficiently as "\Code\MyService\Ready").

```
Scope(\Code, "MyService\Ready")
```

The Scope operation may not find a variable called "MyService\Ready" in Code, but could find a variable called MyService containing a service

object, which itself contains a variable called Ready. This Scope() would return the value of that Ready variable.
If the MyService object does not contain a variable called Ready, but \Code does contain a Ready variable, then Scope will return the value of Code's Ready variable, which may not be what is desired. However, if the ScopeLocal parameter is added and set to TRUE, then it will return Invalid since it will not scope up from MyService to the Ready variable in Code."

ScopeLocal

Note: Deprecated. Do not use in new code.

Description: Exists only for backward compatibility. Use Scope(Module, VarName, TRUE) for all new code.
Performs a scope resolution only if it occurs in the requested context. Returns a reference to the requested member within a module or other object.

Returns: Reference

Usage: Steady State only.

Function Groups: Basic Module

Related to: Scope

Format: ScopeLocal(Object, Member)

Parameters:

Object

Required. Any expression for the object value (module) where Member may be found.

Member

Required. Any text expression for the member name. This must be a simple variable or module. Array references or further scope resolution are not allowed inside the text expression.

Comments: This function differs from the Scope function only in that it

is restricted to the requested module context. It will not search through the scope tree if the Member name is not found in the designated object.
If the final result is Invalid, this function looks for the presence of backslash (\) characters in the second parameter and parses the result.

Example:

```
ScopeLocal(\Code, "TagName");
```

Returns a reference to the given tag object, found within the current module.

SDev

Description: Returns the statistical sample standard deviation for a subsection of an array.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Array, Generic Math

Related to: AMax | AMin | AValid | FiltHigh | FiltLow | FitOffset | FitSlope | Mean | Sum | Variance

Format: SDev(ArrayElem, N)

Parameters:

ArrayElem

Required. Any array element giving the starting point in the array. The subscript for the array may be any numeric expression. If processing a multidimensional array, the usual rules apply to decide which dimension should be used.

N

Required. Any numeric expression giving the number of array elements to use starting at the element given

by the first parameter. If N extends past the upper bound of the lowest array dimension, this computation will "wrap-around" and resume at element 0, until N elements have been processed.

Comments: The function returns an invalid result if either of its parameters is invalid or if there are less than two valid numerical array elements in the specified range. Invalid elements are not included in the calculation.

Example:

```
productStdDev = SDev(weight[0], 5);
```

This computes the standard deviation of elements 0 to 4 of the array weight.

Seconds

Description: Returns the number of seconds since midnight of the current day.

Returns: Numeric

Usage: Script only.
May be used in optimized Tag Parameter Expressions.

Function Groups: Time and Date

Related to: AbsTime | CurrentTime | Now | Today

Format: Seconds()

Parameters: None

Comments: Care must be taken when using this function together with the Today function to determine the current date and time. If the time is within a fraction of a second of midnight, the value of the date may be calculated in one day and the time in the other day giving an apparent error of almost 24 hours. In situations where this is a problem, the AbsTime function may be helpful; however, the best practice is to

use the function CurrentTime.

Examples:

```
If 1 Main;
[
  timeStamp = Seconds();
]
```

This assigns the current time in seconds since midnight to the floating point (or long) variable timeStamp. Another example of how this function could be used is:

```
If 1 Main;
[
  start = Seconds();
  FRead(1, "G:\TOTALS\BARTOTAL.DAT", 120, "%40c", bar4Total);
  duration = Seconds() - start { Compute time for network
  server disk read };
]
```

This example shows how to measure the execution time of various functions in a script. This provides information on how to improve the performance of the application. In this case, if the time is too long, it may be useful to install a disk drive cache such as SMARTDRV.

SectionControl

(System Library)

Description	Creates a control that displays a variable number of sections. Visually, a section consists of a header and content. The control manages the layout and geometry for the sections and runs a caller-supplied module to display the section content (see Comments).
Returns	Object
Usage	Steady State only.
Function Groups	Advanced Module, Graphics
Related to:	DialogInitPos | Droplist | GridList | HScrollbar | Listbox | RadioButtons | SplitList | ToolBar | VScrollbar
Format:	\System\SectionControl(HeadingFont, HeadingTextColor,

HeadingBackColor)

Parameters

HeadingFont

Required. The font to use for the heading.

HeadingTextColor

Required. The color to be used for the heading text.

HeadingBackColor

Required. The color to be used as the background for the heading text.

Comments: As indicated above, a section consists of a header and content. The control manages the layout and geometry for the sections and runs a caller-supplied module to display the section content.
Call a SectionControl in steady-state, wait until the public variable Ready goes TRUE (non-zero), and then call AddSection and DeleteSection (see the Related Sub-functions section above) to add sections to and delete sections from the control.
SectionControl assumes that it is being run in a child window and expands to consume the entire window. It automatically adds a vertical scroll bar when required.

Sub-functions and Variables:

Public Variables:

ExpandEnable

Enable/Disable the expand tool button. Default: 1

WindowEnable

Enable/Disable the new window tool button. Default: 1

PopupTitle

The pop-up window title string. Defaults to the value of Heading.

Height

Height is a public (read-only) variable that gives the overall height of the SectionControl in pixels.

Ready

Ready is a public (read-only) variable that must go to a non-zero value prior to AddSection or DeleteSection being called (see Comments section)

Sub-Functions:

AddSection

Comments: AddSection returns the object value of a SectionControl\DisplaySection instance. This object value is used as a parameter to DeleteSection as well as providing some read-only values that the caller may access. These are:
Ready When 1, the caller can safely access the public variables listed. Default 0.
ExpandEnable Draws and enables the expand tool button. Default 1.
WindowEnable Draws and enables the new window tool button. Default 1.
AddButtons Subroutine module.
Takes a single parameter, being a ToolBar format data array containing additional tool buttons to insert at the LHS of the embedded ToolBar.

Related to:

Format: AddSection(Heading;[, ContentModule, InitiallyExpanded, Param1, Param2, ExpandEnable, WindowEnable, PopupTitle, AvailableHeightPtr])

Parameters:

Heading

Can be a text value or a module value. If a text value, it is rendered in the header. If it is a module value, it is a caller-supplied module to draw the header with the following parameters:

Parameter	Description
X (in)	X coordinate of LHS of drawing area
Y (in)	Y coordinate of the top of the drawing area.
Width (in)	The number of X-pixels available in which to draw.
Height (in)	The number of Y-pixels available in which to draw.
TextColor (in)	The default heading text color.
BackColor (in)	The background color for the heading text.
TextFont (in)	The default heading text font.
Param1 (in)	Param1 passed to AddSection.
Param2 (in)	Param2 passed to AddSection.

ContentModule

A module value of the caller-supplied module to render the content. It is called with the following parameters:

ContentModule	Description
X (in)	X coordinate of LHS of drawing area
Y (in)	Y coordinate of the top of the drawing area.
Width (in)	The number of X-pixels available in which to draw.
ContentHeight (out)	The number of Y-pixels this method needs.
VisibleContent (in)	Boolean: TRUE if content is visible.
Param1 (in)	Param1 passed to AddSection.
Param2 (in)	Param2 passed to AddSection.
Available Height (in)	Numeric value used if the DisplaySection module knows how much height can be made available

The content drawing module must maintain the value of ContentHeight in steady-state. It is used by the control to manage the geometry of other sections in the control. When the Boolean VisibleContent is TRUE (set by the SectionControl), the module must draw the content. When the control sets VisibleContent FALSE, the module must stop drawing the content. By set-

ting the output value ContentHeight, the drawing module tells the control that it is no longer drawing and the section can be collapsed.

InitiallyExpanded

Can be set to TRUE if the heading is to be initially expanded, or FALSE if the heading is to be initially collapsed.

Param1

A parameter passed to the drawing module.

Param2

A parameter passed to the drawing module.

ExpandEnable

A Boolean value giving the initial setting for whether the expand button should be enabled. Default: 1

WindowEnable

A Boolean value giving the initial setting for whether the popup window button should be enabled. Default: 1

PopupTitle

An initial title string for the pop-up window. Defaults to the value of Heading.

AvailableHeightPtr

A numeric value indicating the number of pixels in the Y-axis available to be drawn in. Can be useful where ContentModule needs to present scrollable content to the operator.

DeleteSection

Comments: Deletes a section from the control.

Related to:

Format: DeleteSection(Object)

Parameters:

Object

The object value returned from a previous AddSection call (see above).

Note: The following sub-function, DisplaySection, is not meant to be used externally. It is used internally by AddSection

---

DisplaySection

Comments: Used internally by AddSection.

Related to:

Format: DisplaySection(X, Heading, DisplayModule, VisibleContent, Param1, Param2, EpandEnable, WindowEnable, Pop-upTitle, AvailableHeightPtr)

Parameters:

X

The coordinate of LHS of section.

Heading

The heading text or module for the heading.

DisplayModule

The module value of the content drawing module.

VisibleContent

A Boolean value that can be set to TRUE when content is visible.

Param1

A parameter passed to the drawing module.

Param2

A parameter passed to the drawing module.

EpandEnable

Boolean to enable the Expand button (default: 1)

WindowEnable

Boolean to enable the New Window button (default: 1)

PopupTitle

Text string giving the title of the pop-up window

AvailableHeightPtr

Optional numeric - in case AddSection defines the height available for this section.

SecurityCheck

Security Manager Module

Description: Examines the rules that apply to the current user or the named user to determine if the specified privilege has been granted.

Returns: Boolean

Usage: Script or steady state.

Related to: GetAccountID | GetAccountInfo | GetFullName | GetGroupName | GetUserName | IsLoggedOn | IsSuspended | UIErrorToText

Format: \SecurityManager\SecurityCheck(Privilege [, Suppress, AccountName, DenyCaller, TagName])

Parameters:

Privilege

Required. A privilege number to be used in the security check.

Suppress

Optional. A Boolean indicating whether a privilege refusal dialog should be displayed if the SecurityCheck fails. TRUE to suppress the dialog. Defaults to FALSE.

AccountName

Optional. The name or AccountID of an account against which to check the privilege. Defaults to the

caller's account.

DenyCaller

Optional. The object value of the caller to use when launching the privilege refusal dialog. Defaults to the caller's user session. This is used to control the life-time of the dialog until dismissed by user action.

TagName

Optional. The name of a tag that will be used in the security check.

Comments: This module can be called in script or in steady-state. It returns TRUE if the security check passes and FALSE otherwise. In steady-state the return value may change if the parameters change, if the user's logged on state changes, or if there is a configuration change on the user's account. The security check is iteratively done against each rule in the user account until either a matching rule is found or all rules have been checked and no match found.
If the TagName parameter is Invalid or not specified, SecurityCheck looks in the caller's scope for the nearest tag and uses the name of that tag as the TagName parameter.

Seek

Description: Changes and returns the current position within a stream. The return value is the current stream position after the seek is done.

Returns: Numeric

Usage: Script Only.

Function Groups: Stream and Socket

Related to: BuffStream | CloseStream | FileStream | GetStreamLength | SRead | StreamEnd | SWrite

Format: Seek(Stream, Position[, Mode])

Parameters:

Stream

Required. Any expression that returns the stream to do the Seek on

Position

Required. Any numeric expression that returns either the absolute position, or a relative amount to move the position, depending on Mode. If the mode is relative, this number may be negative (i.e. earlier in the stream).

Mode

An optional numeric expression that denotes which type of positioning to do. If equal to 0, the stream position is changed to Position (absolute mode). If equal to 1, the Position is added to the current stream position (relative mode). Defaults to 0.

Comments: This allows repositioning within a stream.
This function can be used to find the current stream position by setting the last two parameters to "0,1".

Example:

```
If ! Valid(strm);
[
  strm = BuffStream("ABCDEF");
  Seek(strm, 3, 0);
  SRead(strm, "%1c", streamData);
]
```

The variable streamData now contains the string "D".

SelectArea

Description: Selects active graphics statements within a rectangular area in a window.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Graphics

Related to:

Format: SelectArea(Object, Left, Bottom, Right, Top, MustContain)

Parameters:

Object

Required. Any expression for the object value that identifies the window.

Left

Required. Any numeric expression for the left side coordinate of the area.

Bottom

Required. Any numeric expression for the bottom side coordinate of the area.

Right

Required. Any numeric expression for the right side coordinate of the area.

Top

Required. Any numeric expression for the top side coordinate of the area.

MustContain

Required. Any logical expression. If true, an object will be selected only if it is completely enclosed by the area. Otherwise, an object will be selected if any part of that object falls within the area.

Comments: SelectArea is a function that returns the number of objects that have been found and selected within the defined area.

SelectCodePointer

Description: Given a window object and a code pointer for an active graphics object within that window, this function adds the graphics object to the window's selection set.

Returns: Pointer

Usage: Script or steady state.

Function Groups: Graphics

Related to: SelectDAG

Format: SelectCodePointer(Object, CodePointer)

Parameters:

Object

Required. Any expression for the object value that identifies the window containing the graphic.

CodePointer

Required. Any code pointer value expression for the graphic object. (see comments)

Comments: You must first use StatementInstance to obtain the code pointer to be used in the CodePointer parameter.

SelectDAG

Description (i.e. Select Function) This function selects an active graphics DAG.

Returns Code Pointer

Usage Script or steady state.

Function Groups Graphics

Related to: SelectCodePointer

Format: SelectDAG(Statement, Object, Index)

Parameters

Statement

Required. Any expression for the code value of the statement to select.

Object

Required. Any expression for the object value that identifies the module instance where Statement is found.

Index

Required. Any numeric expression for the function within Statement.

SelectGraphic

Description Selects an active graphics statement at a location in a window.

Returns Code Pointer or Steady State

Usage Script Only.

Function Groups Graphics

Related to:

Format: SelectGraphic(Object, X, Y, Dist, N)

Parameters

Object

Required. Any expression for the object value that identifies the window.

X

Required. Any numeric expression for the x-axis coordinate of the selection point.

Y

Required. Any numeric expression for the y-axis coordinate of the selection point.

Dist

Required. Any numeric expression for the maximum distance a graphic may be from (X,Y) and still be considered for selection.

N

Required. Any numeric expression for the object to

select. 0 is the first object, 1 is the next, and so on. This allows selection of graphics that are 'underneath' other graphics.

SelectHandle

Description: Returns a pointer to a handle of selected graphics statements at a location in a window.

Returns: Pointer

Usage: Script or steady state.

Function Groups: Graphics

Related to: DragHandle | SelectHandleNum

Format: SelectHandle(Object, X, Y, DragAll)

Parameters:

Object

Required. Any expression for the object value which identifies the window.

X

Required. Any numeric expression for the x-axis coordinate of the selection point.

Y

Required. Any numeric expression for the y-axis coordinate of the selection point.

DragAll

Required. Any logical expression. If true, dragging a handle will drag all selected handles at (X,Y). Otherwise, only the first selected handle (in the topmost layer) will drag.

Example:

```
If LocSwitch() == 4 GetHandle;
[
  handle = SelectHandle(CurrentWindow(), XLoc(), YLoc(), 1);
```

```
  handle = PickValid(handle, 0);
]
```

This script will be executed when the left mouse button is pressed, and will attempt to grab a valid handle that lies directly under the mouse.

SelectHandleNum

Description: Selects the given handle of selected graphics statements in a window.

Returns: Point

Usage: Script or steady state.

Function Groups: Graphics

Related to: DragHandle | SelectHandle

Format: SelectHandleNum(Object, HandleNum, X, Y)

Parameters:

Object

Required. Any expression for the object value which identifies the window.

HandleNum

Required. Any numeric expression for the handle number which you wish to choose. The handles are numbered as follows.

0 1 2
6 8 7
3 4 5

X

Required. Any variable in which the X screen coordinate for that handle will be returned.

Y

Required. Any variable in which the Y screen coordinate for that handle will be returned.

Comments: This function selects the handle requested in the window requested and returns the user coordinates for that selected handle. The returned coordinates can then be used in the DragHandle function.
Will return invalid for the X & Y values if an attempt is made to select the middle side handles (1, 4, 6, or 7) when those handles are not visible due to the object selection box being too small to display them.

SelectPath

Description: Selects a path given its code pointer value.

Returns: Object

Usage: Script or steady state.

Function Groups: Graphics

Related to:

Format: SelectPath(Object, CodePointer)

Parameters:

Object

Required. Any expression for the object value that identifies the window.

CodePointer

Required. Any expression for the code pointer value of the path to select.

Self

Description: Returns the object value of the current module.

Returns: Object Value

Function Groups: Basic Module

Usage: Script or steady state.

Related to: NParm | Parameter | PType | ResetParm | Return | SystemSelf

Format: Self( )

Parameters: None

Comments: This is the only way to get a module's object value, and is commonly used in functions such as NParm, Parameter, ResetParm, and Return.

Example:

```
modPtr = FindVariable("MyChildMod", Self(), 0, 1);
```

This function searches for the module called MyChildMod starting at the current module and proceeding through its ancestral tree.

Send

(RPC Manager Library)

Description: This subroutine sends a message by invoking a remote procedure call (RPC).

Returns: Current session id of remote procedure, else RPC_NO_SID

Usage: Script Only.

Function Groups: Network

Related to: ConnectToMachine | DisconnectFromMachine | GetServer | GetServersListed | GetStatus | IsClient | IsPotentialServer | IsPrimaryServer | Register (RPC Manager) | SetRemoteValue

Format: \RPCManager\Send(Service, RemoteGUID, ModeCutOff, SendServer, MachineName, SendClients, ExecLocally, Recursive, ModuleName, ModuleContext, UpdateObject, InputSessionID [, Parmeters…])

Parameters:

ServiceName

Required. The name of the service to transmit to. For directed RPCs, set this value to either Invalid or a zero-length string.

RemoteGUID

Required. The GUID of the application to receive the RPC. If Invalid, RemoteGUID is searched for in the caller's scope.

ModeCutOff

Required. The service synchronization mode above which this message should not be sent. Normal RPCs should set this to RPC_ACCEPT_ALL mode.

SendServer

Required. If set to "1", this flag will transmit this RPC to the service instance that is currently the server for the service. Ignored for directed RPC requests.

MachineName

Required. The IP or name of the workstation to be used for a directed RPC. Invalid for service RPCs.

SendClients

Required. If set to "1", this flag will transmit this RPC to all service instances that are currently clients of the service server. Ignored for directed RPC requests.

ExecLocally

Required. If set to "1", this flag forces the RPC to be executed locally. Used with directed RPCs.

Recursive

Required. If set to "1", and SendClients is also set to "1", this flag will transmit this RPC to all service instances that are clients of this workstation and all service instances that are clients of them.
If set to a "1" and SendServer is also a "1", will transmit to servers of this workstation and servers of those

workstations. This is of use when "clients of clients" are configured.

In most cases, it is wise to set this flag when making service broadcast updates.

ModuleName

Required. The textual name of the RPC subroutine to be executed. Must be valid.

ModuleContext

Required. The context in which the "ModuleName" will be executed. The "base" context for a VTScada layer-based application is "\Code".
For a non-VTScada (pure script) application, the base context is the root of the application specified by the RemoteGUID parameter. Must be valid.

UpdateObject

Required. If valid, is an object that will act as a holding point for the RPC until it is actually transmitted to the remote workstation. A subsequent RPC with the same UpdateObject value will discard the previous RPC if it has not yet been transmitted, and replace it with the new one. This is of most use to services whose updates completely negate the effect of previous updates, and serves to minimize the transmission of redundant RPCs.

InputSessionID

Required. If Invalid, the RPC will be queued for transmission. If valid, will be interpreted as a SID which must match the current SID for the remote application. Otherwise, the RPC will not be queued for transmission.
Only of use in directed RPCs.

Parameters

A set of up to 32 parameters to the RPC subroutine. Can be any mixture of the legal types. Supplying a parameter of an illegal type will cause it to be replaced with Invalid when the RPC subroutine is invoked.

Comments: This subroutine is a member of the RPC Manager's Library, and must therefore be prefaced by \RPCManager\, as shown in the "Format" section. If the application you are developing is a script application, the subroutine call must be prefaced by System\RPCManager\, and the System variable must be declared in AppRoot.src.
The method returns the current SID of the remote application if the message was queued for transmission. Otherwise, it will return RPC_NO_SID.

Example:

```
If 1 Main;
[
  sessID = \RPCManager\Send("MyService" { service }, \RemoteGUID,
             \RPC_ACCEPT_FILTER { mode cut-off },
             1 { server }, Invalid { machine },
             0 { clients }, 0 { locally },
             0 { recursive }, "Start" { module },
             "\" { service scope },
             Invalid { queue msgs },
             Invalid { no initial session ID },
             { Parameters: } rev, myName);
 sessID = \RPCManager\Send("MyService" { service },
             \RemoteGUID,
             \RPC_ACCEPT_FILTER { mode cut-off },
             1 { server }, Invalid { machine },
             0 { clients }, 0 { locally },
             0 { recursive }, "Finish" { module },
             "\" { service scope },
             Invalid { queue msgs },
           sessID { original server's sess ID },
             { Parameters: } index);
]
```

This will cause the modules called Start and Finish, which are found in the scope of the service called MyService, to be executed on the server. Note that by handing in the session ID from the first call to the second call it is guaranteed that module Finish will either be queued for the workstation that was the server at the time when the first message was

queued or will be discarded. It will not be sent to another workstation that may take over servership for the MyService service.

Related Information:

Refer also to "RPC Manager Service" for a listing of Service Control Methods, RPC Methods, and Deprecated RPC Methods.

SendMail

Description: Sends a string to an email server using the Simple Mail Transport Protocol (SMTP)

Returns: Nothing

Usage: Script Only.

Function Groups: Email

Related to: ValidateEmailAddrs

Format: \SendMail(Server, To, From, Subject, Message, Error [,Attachments, AttachmentStreams, OptionalHeaders, ErrorText, SenderMailbox, Username, Password, UseTLS, Bcc, Charset, Port])

Parameters:

Server

Required. Any text or buffer expression of the mail server name or IP address

To

Required. Any text or buffer expression of the address of destination

From

Required. Any text or buffer expression of the address of the person from whom the mail is from

Subject

Required. Any text or buffer expression of the subject of the message

Message

Required. Any text or buffer expression of the message – no attachments

Error

Required. A pointer to an error variable, set to 0 if OK. The Error parameter should initially be set Invalid by the caller. When this method is done, it will set Error to one of the following values:

Error	Meaning
0	Mail has been successfully sent.
1	Unable to open a connection to the server.
2	Server did not send a good SMTP welcome message.
3	Server rejected SMTP HELO message.
4	Server rejected SMTP MAIL message.
5	Server rejected SMTP RCPT message.
6	Server rejected SMTP DATA message.
7	Server rejected message body.
8	Badly formatted TO: address.
9	Server does not support AUTH.
10	Server does not support implemented AUTH. mechanisms.
11	Failure to login to SMTP server.
12	TLS wanted for SMTP, but not available
13	TLS unavailable or negotiation with server failed
14	Badly formatted bcc address

Attachments

An optional parameter that is an array of file names to be attached to the email message.

AttachmentStreams

An optional parallel array of attachment streams.

OptionalHeaders

A string of optional MIME headers, separated by CRLF.

ErrorText

An optional pointer to the variable that holds the error text.

SenderMailbox

An optional parameter which is the SMTP reverse path. SenderMailbox overrides the From parameter. This parameter may be used as an alternative to the From parameter when specifying the "MAIL FROM" address in the SMTP.

The From parameter would still be used when specifying the "From:" address in the email header. SenderMailbox should be used only when an email is intended to look like it came from a particular address, but has actually originated from another address.

Username

Allows a user name to be provided to email servers that require authentication.

Password

Allows a password to be provided to email servers that require authentication.

UseTLS

Boolean to indicate that transport layer security should be used.

Bcc

Optional list of recipients, who should receive a bcc

copy of the message.

Charset

Optional. Text string to be used for MIME character set.

Port

Optional numeric. Use to specify which port the SMTP server should use. If not provided, the value of the SMTPPort application setting is used, defaulting to 25.

Comments: This function opens a TCP/IP socked on port 25 (or other if specified) of the specified mail server. The Message is then sent using the SMTP protocol.
If both Username and Password are valid, then SendMail will attempt to login to the mail server before sending the email message. If the server does not support authentication, then SendMail will abort with an error message
If both Username and Password are valid, but the server does not support PLAIN authentication, then SendMail will abort with an error.
While PLAIN *must* be available, some server configurations require that a TLS connection (aka SSL) to be negotiated before allowing PLAIN. VTScada does not support authentication over TLS. It is assumed that communication between the email Server and the VTScada client will be over a secure network such as an intranet.
If either Username or Password is missing or invalid, then the message will be sent without a login attempt.
The To: address will accept a string that conforms to the address-list ABNF from RFC 5322.

Example:

```
\SendMail("192.168.0.201",
          "sales@trihedral.com",
          "vts@trihedral.com",
          "Great Product!",
          "VTScada is Great\n\rMust buy lots!",
          &ErrorCode);
```

SerBreak

Description: Sends a break character to a serial port.

Returns: Nothing

Usage: Script Only.

Function Groups: Serial Port

Related to: COMPort | SerCheck | SerialStream

Format: SerBreak(Port, Status)

Parameters:

Port

Required. Any numeric expression for the serial port number (opened with a ComPort function) or serial stream value (returned from a SerialStream function).

Status

Required. Any logical expression. If true (non-0), a break will be sent to the serial port defined in Port, if false (0), the break will be cleared from the serial port.

Example:

```
If MatchKeys(1, MakeBuff(1, 27) { <ESC> key pressed }) BreakSent;
[
  SerBreak(4, 1);
]
```

The above code causes a break to be sent to serial port 4 if the user presses the <ESC> key.

SerCheck

Description: Check Serial Port. This function returns the immediate or cumulative serial port status.

Returns: Numeric

Usage: Script Only.

Function Groups: Serial Port

Related to: COMPort | SerBreak | SerialStream

Format: SerCheck(Port, Reset)

Parameters:

Port

Required. Any numeric expression for the serial port number (opened with a ComPort function) or serial stream value (returned from a SerialStream function).

Reset

Required. Any logical expression. If true (non-0), the serial port status register will be cleared.

Comments: The return value consists of 9 bits that indicate the status of the following

Bit	Meaning
0	Clear To Send (CTS)
1	Data Set Ready (DSR)
2	Ring Indicator (RI)
3	Carrier Detect (CD)
4	Overrun error
5	Parity error
6	Framing error
7	Break signal detected
8	Receive buffer overflow (always cumulative)

Example:

```
If 1 Send;
[
  parityErr = Cond(And(SerCheck(2, 0), 0b00100000), 1, 0);
  IfThen(parityErr, ForceState("Retry"));
]
```

In this script, port 2 is checked for a parity error. If there is one, a state transfer to state Retry will occur, otherwise the destination state in the script's trigger (state Send) will be active next.

SerialNum

Description: Returns the serial number of the copy of VTScada running.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Software and Hardware

Related to: CommandLine | Version

Format: SerialNum()

Parameters: None

Comments: This function can be used to provide copy protection for user-developed modules by checking for a specific copy of VTScada before performing the desired task.

Example:

```
ZText(20, 20, Cond(SerialNum() != 2975, "Unauthorized serial
number!", ""), 15, 0);
```

This displays a message if the serial number of VTScada is not 2975.

SerialStream

Description: Returns a serial stream that can be used in any of the serial port functions or with any of the stream functions. Please note that the ComPort function (which functions somewhat differently than the SerialPort function) may also be util-

ized.

Returns: Stream

Usage: Script Only.

Function Groups: Serial Port, Stream and Socket

Related to: COMPort | SerCheck | SerIn | StrLen | SerOut | SerRcv | SerSend | SerString | SerStrEsc | SerWait

Format: SerialStream(Port, ReceiveLen, TransmitLen, Baud, DataBits, StopBits, Parity, RTS, XOnXOff, Control)

Parameters:

Port

Required. Any numeric expression giving the serial port number to be used. For COM1: Port = 1. For COM2: Port = 2. The valid range for Port is 1 to 4096.

ReceiveLen

Required. Any numeric expression giving the size of the receiver buffer in bytes. ReceiveLen must be in the range 2 to 32766.
If more bytes are received than can fit in the receive buffer before your application removes them using SerRcv or a similar VTScada function, the additional data will be lost.

TransmitLen

Required. Any numeric expression giving the size of the transmitter buffer in bytes. TransmitLen must be in the range 2 to 32766. The buffer must be large enough to hold the maximum number of bytes pending transmission at any instance.

Baud

Required. Any numeric expression giving the baud rate. Baud must be in the range 10 to 57600, and must divide evenly into 115200 with no more than 2.5%

error.

DataBits

Required. Any numeric expression giving the number of data bits per character. DataBits must be 5, 6, 7, or 8.

StopBits

Required. Any numeric expression giving the number of stop bits per character. StopBits must be 1 or 2.

Parity

Required. Any numeric expression giving the parity checking to use:

Parity	Checking
0	No parity
1	Odd parity
2	Even parity
3	0 Stick (space parity)
4	1 Stick (mark parity)

RTS

Required. Any numeric expression that gives the RTS buffer control method. RTS is on while transmitting. When a transmission is complete, RTS is off.
This is usually used to control the transmitters on RS-422/485 ports. This parameter has no effect if the automatic RTS control is selected in the Control para-

RTS	Buffer Control Method
0	Force RTS off
1	Force RTS on
2	Half-duplex operation
3	Controlled by SerRTS function

Acceptable values of the RTS parameter are as follows:

RTS	Buffer Control Method
0	Force RTS off
1	Force RTS on
2	Half-duplex operation
3	Controlled by SerRTS function

If this parameter is 2, the SerRTS function can set its value. However, regardless of SerRTS, the RTS control line will be asserted when data is sent.

If the SerRTS is called to change the RTS line while data is being transmitted, the RTS line will not change when the last byte is sent. If SerRTS is not executed while the data is transmitted, the RTS line will be cleared after the last byte is transmitted.

XOnXOff

Required. Any logical expression. If true (non-0) software flow control is to be used. If false (0) it is not.

Control

Required. Any numeric expression that specifies the handling procedure for the clear to send (CTS), carrier

Control	Bit No.	Handling Procedure
1	0	DTR on (otherwise DTR is off)
2	1	Enable CTS control
4	2	Enable CD control
8	3	Enable DSR control
16	4	Enable RTS/CTS control
32	5	Enable DTR/DSR control

If bit 1 (CTS control) is set, data will only be transmitted if the CTS signal is on. If CTS control is disabled, the CTS line is ignored.
If bit 2 (CD control) is set, data will only be transmitted when the CD signal is on. If CD control is disabled, the CD line is ignored.
If bit 3 (DSR control) is set, data will only be transmitted when the DSR signal is on. If DSR control is disabled, the DSR line is ignored.
If bit 4 (RTS/CTS control) is set, the CTS control behaves as described above, and the RTS line will be held high until the receive buffer reaches 75% full. It will then go low, indicating to the other device to stop transmitting data. The RTS line will go high again when the receive data buffer drops below 25% full. This is known as hardware flow control. RTS/CTS control enabled overrides the RTS parameter.
If bit 5 (DTR/DSR control) is set, the DSR control behaves as described above, and the DTR line will be held high until the receive buffer reaches 75% full. It will then go low, indicating to the other device to stop transmitting data. The DTR line will go high again when the receive data buffer drops below 25% full. This is known as hardware flow control. DTR/DSR control enabled overrides bit 0, DTR on option.

Comments: The parameters for this function are a reduced set from the Comport statement.
Make sure that VTScada's mouse (if it is serial) is on a different port, because the mouse and SerialStream can interfere. Also make sure that no

other hardware or software is interfering with the serial port hardware interrupts (IRQ4 for COM1:, IRQ3 for COM2:). Network cards often use IRQ4, which will cause a problem with a mouse or SerialStream on COM1:.
The stream will automatically close when there are no variables referencing the stream. However, if there are bytes still in the transmit buffer, they will not be sent before the stream is closed.

Example:

```
streamCom2 = SerialStream(2 { COM2: },
                     1024 { Buffer 1024 bytes of received data },
                     1024 { Buffer 1024 bytes of transmitted data },
                     9600 { Baud rate },
                     8 { Data bits per byte },
                     1 { Stop bit per byte },
                     0 { No parity bit },
                     1 { Force RTS on },
                     0 { No Software flow control (XonXOff) },
                     3 { Control: DTR On, CTS control enabled });
```

This opens COM2: for use with serial port functions. These functions should use streamCom2 as their Port parameter.

SerIn

Description: Get Serial Port Byte. This function returns the next byte in the receive buffer.

Returns: Byte

Usage: Script Only.

Function Groups: Serial Port

Related to: COMPort | SerCheck | StrLen | SerOut | SerRcv | SerRTS | SerSend | SerStrEsc | SerString | SerWait

Format: SerIn(Port, Peek)

Parameters:

Port

Required. Any numeric expression for the serial port number (opened with a ComPort function) or any stream value.

Peek

Required. Any status expression to control whether the byte being read should also be removed from the receive buffer.
If Peek is true, the byte is not removed from the receive buffer.

Comments: If no byte is available, the return value is invalid.

Example:

```
If 1 Continue;
[
  chkSum = SerIn(2, 0);
  IfThen(! Valid(chkSum), ForceState("Wait"));
]
```

This reads one byte from the receive buffer for serial port 2. If none is available, chkSum will be invalid. Notice that the next state to become active is dependent on whether or not a byte has been successfully read.

SerLen

Description: Serial Port Buffer Length. This function returns the number of bytes currently in the receive or transmit buffers.

Returns: Numeric

Usage: Script Only.

Function Groups: Serial Port, String and Buffer

Related to: COMPort | SerCheck | SerIn | SerOut | SerRcv | SerRTS | SerSend | SerStrEsc | SerString | SerWait

Format: StrLen(Port, Option)

Parameters:

Port

Required. Any numeric expression for the serial port number (opened with a ComPort function) or any stream value.

Option

Required. Any status expression. If Option is true, the number of bytes in the transmit buffer is returned. If Option is false, the number of bytes in the receive buffer is returned.

Comments: This function may only appear in a script.

Examples:

```
If ! Valid(rcvBufLen);
[
   rcvBufLen = SerLen(2, 0);
]
```

This finds the number of bytes presently in the receive buffer on serial port 2. This will always be a valid number equal to 0 or greater.

SerOut

Description: Send Serial Port Byte. This statement sends a byte to the transmit buffer.

Returns: Nothing

Usage: Script Only.

Function Groups: Serial Port, String and Buffer

Related to: COMPort | SerCheck | SerIn | StrLen | SerRcv | SerRTS | SerSend | SerStrEsc | SerString | SerWait

Format: SerOut(Port, Value)

Parameters:

Port

Required. Any numeric expression for the serial port number (opened with a ComPort function) or any

stream value.

Value

Required. Any numeric expression which gives the byte value to send. Value must be in the range 0 to 255.

Comments: The byte will not be sent if the transmit buffer is full.

Example:

```
If MatchKeys(2, "t");
[
  SerOut(2, Cond(chkSum == calcChksum, 6, 21);
]
```

This waits until the operator presses the "t" key, then it compares chkSum to calcChksum. If equal, a 6 byte (ASCII ACK) is sent. If not equal, a 21 byte (ASCII NAK) is sent. The serial port used is 2.

SerRcv

Description: Serial Port Receive. This function returns a buffer containing a string read from the receive buffer.

Returns: Buffer

Usage: Script Only.

Function Groups: Serial Port

Related to: COMPort | SerCheck | SerIn | StrLen | SerOut | SerRTS | SerSend | SerStrEsc | SerString | SerWait

Format: SerRcv(Port, Count)

Parameters:

Port

Required. Any numeric expression for the serial port number (opened with a ComPort function) or any stream value.

Count

Required. Any numeric expression, which gives the number of bytes to read. Value must be in the range 0 to 65500.

Comments: The return value is a buffer containing Count bytes, unless fewer bytes were available in the receive buffer. This function may be used to flush the serial port receive buffer.

Example:

```
If SerWait(2, 12) { Wait for 12 bytes buffered for port 2};
[
  data = SerRcv(2, 12);
]
```

This waits until 12 bytes are in the receive buffer for serial port 2, and then reads them into the variable data. Byte 0 of data will be the oldest byte in the receive buffer (received first by VTS).

SerRTS

Description: Sets or clears the RTS line on a serial communication port.

Returns: Nothing

Usage: Script or steady state.

Function Groups: Serial Port

Related to: COMPort | SerCheck | SerialStream | SerIn | StrLen | SerOut | SerRcv | SerSend | SerStrEsc | SerString | SerWait

Format: SerRTS(Port, Level)

Parameters:

Port

Required. Any numeric expression for the serial port number (opened with a ComPort function) or serial stream value (returned from a SerialStream function).

Level

Required. TRUE if the RTS line is to be turned on or

false if the RTS line is to be turned off.

Comments: The corresponding ComPort statement must have the RTS parameter set to 2 or 3.

Example:

```
RTS = 0;
...
ComPort(2 { COM2: },
        1024 { Buffer 1024 bytes of received data },
        1024 { Buffer 1024 bytes of transmitted data },
        9600 { Baud rate },
        8, 1 { Data bits/byte, stop bits/byte },
        0 { No parity bit },
        3 { RTS may be controlled by SerRTS function },
        0, 0, 0 { Obsolete parameters },
        3 { Control: DTR On, CTS control enabled },
        0, 0, 0, 0, 0, 0, 0, 0, 0 { Obsolete parameters });
If MatchKeys(2, "R");
[
  RTS = 1 - RTS { Toggles RTS between 1 and 0 };
  SerRTS(2, RTS);
]
```

Comport 2 is opened to allow the setting of the RTS line. The second statement then toggles the RTS line of serial port 2 on and off every time the letter R is pressed on the keyboard.

SerSend

Description: Serial Port Send. This function writes a string to the transmit buffer and returns the number of bytes written.

Returns: Numeric

Usage: Script Only.

Function Groups: Serial Port

Related to: COMPort | SerCheck | SerIn | StrLen | SerOut | SerRcv | SerRTS | SerStrEsc | SerString | SerWait

Format: SerSend(Port, Buffer, MaxBytes, Escape)

Parameters:

Port

Required. Any numeric expression for the serial port number (opened with a ComPort function) or any stream value.

Buffer

Required. Any text expression to send.

MaxBytes

Required. Any numeric expression which gives the maximum number of bytes to send.

Escape

Required. Any numeric expression which gives the byte value of an escape code. Whenever an escape code is encountered, two escape codes will be transmitted.

To be valid, Escape must be in the range of 0 to 255. If the Escape parameter is greater than 0 or less than 256, it is transmitted with escape codes.

If Escape is a value greater than 255, it will perform in the same manner that it would if Escape were a value of 0.

If the value of Escape is less than 0, it doesn't take into account escape codes and the buffer is transmitted in its entirety, stopping at MaxBytes.

Comments: The return value is the actual number of buffer bytes successfully placed in the transmit buffer (not including extra escape codes). MaxBytes should be transmitted, unless the end of the buffer is encountered, or a transmit buffer overflow occurs.

Example:

```
If MatchKeys(2, "S");
[
  SerSend(2, "Hello World ", 12, 0);
]
```

The code above sends the message "Hello World" through serial port 2.

```
if (escParm < 0) {
  stream->write(xmitbuff, i);
  sent = i;
}
else {
  sent = 0;
  while (sent < i) {
    count = 0;
    while (sent + count < i &&
      (W_SHORT) ((W_UCHAR) xmitbuff[sent + count++]) != esc);
    stream->write(&(xmitbuff[sent]), count);
    sent += count;
    if (xmitbuff[sent - 1] == esc)
      /*** Must be an ESC character - low byte must be first ***/
      stream->put(esc);
  }
}
```

In the above example, i is either the Buffer Length or MaxBytes, whichever is smaller.

SerStrEsc

Description: Serial Port Receive With Escape. This function reads the receive buffer until a specified character (the 'escape' character) is encountered, incrementing an offset counter for each character read that is not the Escape character. It is assumed that, where the message contains the Escape character as part of the message, that character is 'escaped' by being doubled.

Returns: Numeric

Usage: Script Only.

Function Groups: Serial Port

Related to: COMPort | MakeBuff | SerCheck | SerIn | StrLen | SerOut | SerRcv | SerRTS | SerSend | SerString | SerWait

Format: SerStrEsc(Buffer, Offset, Port, Escape)

Parameters:

Buffer

Required. Any text variable to write the received data. This text buffer must already exist. It could be created

by another function such as MakeBuff, or by assignment of a text constant.

Offset

Required. Any numeric expression which gives the offset from zero where SerStrEsc will start writing data to Buffer. If Offset is negative, Offset is returned and nothing is done.

Port

Required. Either a numeric expression for the serial port number (opened with a ComPort function) or any stream value.

Escape

Required. Any numeric expression which gives the terminating character code. This must be in the range 0 to 255 to be valid. A value of -1 means that no such code exists.

Comments:

Data is read until the receive buffer is empty, Buffer is full, or an Escape value is encountered. If two successive Escape values are received, one is written to Buffer and reading continues normally. Otherwise, reading is terminated if the character following the Escape is not another Escape character. This is useful for reading the serial port receive buffer where the end of a message is signaled by a particular byte sequence. The byte following the final Escape is placed in the buffer.
The value returned by the function increases with each character read. When a single instance of the Escape value is encountered, the return value is the negative of the final offset before reading in the Escape character (the Escape character is discarded). The return value can be used in successive SerStrEsc calls as the Offset parameter to fill Buffer until a Escape character is encountered.

Example:

```
response = MakeBuff(1, 32) { Fill the buffer with spaces };
...
  { Read port until DLE encountered }
If (! Valid(pos) || pos >= 0) && SerWait(2, 1) { Variable becomes neg-
ative when <DLE> encountered };
[
  pos = SerStrEsc(response              { Buffer },
                  PickValid(pos, 0)    { Start at pos or 0 },
                  2                     { Port number },
                  0X10                  { Data end code (<DLE> or
decimal 16) });
]
  { DLE found, all done }
IF Pos < 0  ProcessResponse;
[
   { calculate the size of the response message }
   RespSize = pos * -1;
]

ZText(10, 10, response, 0, 0);
```

This reads all characters from serial port 2 until a single byte 0x10 is encountered. The byte following the escape code will be the final character in response. When the escape is received, pos will become a pointer to where the last byte was written into the buffer, and the ZText statement will display the received message. Suppose the following (decimal) bytes were received by VTScada on serial port 2:

```
1, 2, 3, 16, 16, 4, 5, 16, 3, 6, 9, 8
```

The response would contain the following:

```
1, 2, 3, 16, 4, 5, 3
```

and the following bytes would be left in the receive buffer:

```
6, 9, 8
```

SerString

Description:	Serial Port String Receive. This function reads the receive buffer until a string is encountered and returns the final offset in the buffer.
Returns:	Numeric
Usage:	Script Only.
Function Groups:	Serial Port, String and Buffer

Related to: COMPort | MakeBuff | SerCheck | SerIn | StrLen | SerOut | SerRcv | SerRTS | SerSend | SerStrEsc | SerWait

Format: SerString(Buffer, Offset, Port, String)

Parameters:

Buffer

Required. Any text buffer to write the received data to. It must already exist – either by creating it with a function such as MakeBuff, or assignment of a text constant.

Offset

Required. Any numeric expression which gives the offset from 0 where SerString will start writing data to Buffer.

Port

Required. Any numeric expression for the serial port number (opened with a ComPort function) or any stream value.

String

Required. Any text expression which gives the terminating string.

Comments: Data is read until the receive buffer is empty, Buffer is full, or String is encountered. This is useful for reading the serial port receive buffer where the end of a message is signaled by a particular byte sequence. The optional return value is the final offset after reading into Buffer, unless String was encountered, or Offset is negative.
If String was encountered, the return value is the negative of the final offset after reading in the String sequence. If Offset is negative, Offset is returned and nothing is done. The return value can be used in successive SerString calls as the Offset parameter to fill Buffer until String is encountered.

Example:

```
If (! Valid(pos) || pos > 0) && SerWait(2, 1)
{ Pos is positive until "END" };
[
  pos = SerString(response { Buffer },
  Cond(Valid(pos), pos, 0) { Start at the
  beginning, do successive reads },
  2, "END" { Port no. & terminating string });
]
```

This reads from serial port 2 until the string "END" is encountered. As the message is read from the serial port, pos will be StrLen(response) and "END" will be the final character sequence in response - when it is received, pos will become -StrLen(response).

ServerList

Description: Returns a pointer to an array of all available servers visible from this workstation.

Returns: Pointer

Usage: Script Only.

Function Groups: Network

Related to: TServerList | WKStaInfo

Format: ServerList(Obsolete, Domain, Obsolete)

Parameters:

Obsolete

Parameter is no longer used, but is maintained for backward compatibility with previous versions of VTScada. Set to 0.

Domain

Required. Any text expression for the domain. If this is invalid, the current domain will be used.

Obsolete

Parameter is no longer used, but is maintained for backward compatibility with previous versions of

VTScada. Set to 0.

Comments: This function will return the resultant array of servers, or if network problem is encountered or Domain is not found on the network, an error code will be returned.

Example:

```
If ! Valid(serverArray);
[
  serverArray = ServerList(0, WkStaInfo(2), 0);
]
```

The second parameter in the ServerList statement is included as an added example of how many functions work together - in actual fact it is redundant, since it simply designates Domain as being the current domain, which could also have been done by simply entering an invalid.

ServerSocket

Description: Returns a server WinSock socket stream given a handle returned by a SocketServerStart function or an integer error code.

Returns: Stream

Usage: Script Only.

Function Groups: Network, Stream and Socket

Related to: ClientSocket | CloseStream | SocketAttribs | SocketServerEnd | SocketServerStart | SocketWait | SRead | SWrite | TCPIPReset

Format: ServerSocket(Handle)

Parameters:

Handle

Required. Any numeric expression for the handle returned by the SocketServerStart function.

Comments: This function is used as part of a WinSocket-compliant

server application. First, start a socket server using SocketServerStart. Then, use the SocketWait function as the trigger for an action script. Use this function in the script to connect a socket to the client application that triggered the SocketWait function. If the socket connection is lost (client shutdown) the stream is closed and set to a value of 0 (no error code returned).

If you experience difficulty with TCP/IP, a useful troubleshooting tool is the Windows™ diagnostic "NetStat.exe", used to display information about the network. Also of use is the Windows™ "Ping.exe" diagnostic which can be used to test the hardware connection. These files are normally found in the Windows™ directory. Consult the Windows™ documentation on their usage.

Example:

```
Init [
  If 1 Wait;
  [
    sHandle = SocketServerStart(0 { TCP/IP },
    20000{ Port number offered },
    1024 { Transmit buff length },
    1024 { Receiver buff length },
    1 { No transmit delay });
  ]
]
Wait [
  If SocketWait(sHandle) Main { Wait for client to connect };
  [
    server = ServerSocket(sHandle);
  ]
]
Main [
  If GetStreamLength(server) > 0 || MatchKeys(2, "r");
  [
    SRead(server, Concat("%", Concat(GetStreamLength(server),
    "c")), Data);
    SWrite(server, "%s", Data);
  ]
  { Always close socket when complete !!! }
  If WindowClose(Self());
  [
    CloseStream(server);
    SocketServerEnd(sHandle);
    Slay(Self(), 1);
  ]
  { Display data from client and server status }
```

```
  ZText(0, 50, Data, 0, 0);
  ZText(0, 100, Cond(ValueType(server) == 8 ,
  "Connected",
  "Not Connected"),
  10, 0);
]
```

SerWait

Description: Serial Port Wait. This function returns true when the receive buffer is a specified length.

Returns: Boolean

Usage: Steady State only. See: Rules for Usage.

Related to: COMPort | SerCheck | SerIn | StrLen | SerOut | SerRcv | SerRTS | SerSend | SerStrEsc | SerString

Format: SerWait(Port, Count)

Function Groups: Serial Port

Parameters:

Port

Required. Any numeric expression for the serial port number (opened with a ComPort function) or any stream value.

Count

Required. Any numeric expression, which gives the number of bytes to wait for. Value must be in the range 0 to 65500.

Comments: The return value is true when at least Count or more bytes have been received from Port. This is treated as a high priority function, and anything using the return value of SerWait will be executed as soon as Count or more bytes are received from Port.

Example:

```
If SerWait(3, 16);
[
```

```
  recBuff = SerRcv(3, 16);
]
```

This action trigger will be true when 16 or more bytes are in the receive buffer from port 3, at which point the script will execute and the bytes in the receive buffer will be read into the buffer called recBuff.

SetAllBlocks

(RPCManager Library)

Description: This subroutine executes on the client. It accepts all of the blocks and data for a service.

Returns: Nothing

Usage: Script Only.

Function Groups: Network

Related to: COMPort

Format: SetAllBlocks(NDrivers, Buffer, Vals, TimeStamp, Attrib)

Parameters:

NDrivers

Required. Any number of drivers in the Buffer.

Buffer

Required. A stream containing all of the data.

Vals

Required. A packed array of driver values.

TimeStamp

Required. A packed array of driver values.

Attrib

Required. A packed array of driver values.

Comments: SetAllBlocks executes on the client. It is called from GetServerChanges on the server via RPC. SetAllBlocks accepts all of the blocks and data for a service.

SetBit

Description: Sets or clears a specific bit in a value and returns the result.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Bitwise Operation

Related to: Bit

Format: SetBit(Value, BitNumber, Option)

Parameters:

Value

Required. Any numeric expression that gives the number to modify the bit in.

BitNumber

Required. Any numeric expression giving the bit number to set or clear. Bit 0 is the least significant bit. Legal values are from 0 to 31 inclusive.

Option

Required. Any status expression indicating whether the bit is to be set or cleared in the Value. A true indicates that the bit is to be set (i.e. set to 1), and a false indicates that it is to be cleared (i.e. set to 0).

Comments: This function is useful for saving a series of status values in a single short or long variable. If any argument is invalid, the return value is invalid.

Examples:

```
a = SetBit(0, 1, 1);
```

In this simple example, the values for a will be 2 { 0b00000010 }. This is not a particularly useful example of SetBit, though. Most likely you will want to use it in a statement like the following:

```
b = SetBit(b, 2, 1);
```

In this case, bit number 2 will be set to 1. It is important to note, however, that if b is invalid, this statement will have no effect (i.e. it will not make b valid). A safer way to accomplish the setting of bit 2, then, might be to write

```
b = SetBit(PickValid(b, 0), 2, 1);
```

so that the b being invalid case is covered.

SetByte

Description: Writes a single byte to a buffer.

Returns: Nothing

Usage: Script Only.

Function Groups: String and Buffer

Related to: GetByte | MakeBuff

Format: SetByte(Buffer, Offset, Value)

Parameters:

Buffer

Required. Any buffer expression giving the buffer to set. This buffer must already exist. It could be created by a function such as MakeBuff, or by assignment of a text constant.

Offset

Required. Any numeric expression giving the offset from the start of the buffer in bytes, starting from 0.

Value

Required. Any numeric expression giving the byte value to write at Offset bytes from the beginning of Buffer.

Comments: This statement may only appear in a script.

Example:

```
msg = "ABCDEF";
If MatchKeys(2, "g");
[
  SetByte(msg, 1, 65);
]
```

The value of msg will be "AACDEF" when the user presses "g" on the keyboard.

SetClock

Description: Sets the VTScada system clock and calendar.

Returns: Nothing

Usage: Script Only.

Function Groups: Time and Date

Related to: Seconds | Today

Format: SetClock(Date, Time)

Parameters:

Date

Required. Any numeric expression giving the new date, in days since 1 January 1970.

Time

Required. Any numeric expression giving the new time, in seconds since midnight.

Comments: This statement allows the synchronization of the VTScada system clock/calendar with another clock/calendar. All pending timers, and all built-in statements and functions that depend on the system clock/calendar are adjusted to match. Caution is necessary, because VTScada will not adjust times and dates stored in variables or files (VTScada doesn't know if a variable contains a number or time or date). I/O drivers which time and date stamp results will not adjust themselves retroactively. However, as each new

action occurs, the new time and date will be used. This is because I/O drivers are not built-in functions, they are modules.

Example:

```
If ZEditField(10, 40, 110, 10, minPastHr, 2, 1, 1);
[
  currTime = Seconds();
  minPastHr = Cast(minPastHr, 0){ Type status (0 - 255) };
  minPastHr %= 60 { Valid range is from 0 - 60 };
  convTime = currTime { Seconds since midnight }
  - currTime % 24 { Seconds past the hour }
  + (minPastHr * 60){ New seconds past the hour };
  SetClock(Today(), convTime) { Set the system clock };
]
```

This takes a value that has been entered into an input field and sets the system clock by it. The date and hour remain unchanged, only the minutes are set.

SetCodeText

Description: Will modify a source code file to replace the text for a given CodeValue with the new text.

Returns: Invalid on success, numeric error code otherwise.

Usage: Script Only.

Function Groups: Advanced Module, File I/O

Related to:

Format: SetCodeText(CodeValue, NewText)

Parameters:

CodeValue

Required. Any code value giving the statement whose text is to be replaced..

NewText

Required. Any text or a buffer

Comments: The given source file must be present and have read/write

privileges. This function does not immediately update the statement's currently loaded module until that module is recompiled (usually upon a shutdown/restart cycle.

SetCursor

Description: Sets the mouse cursor type for the window.

Returns: Nothing

Usage: Script Only.

Function Groups: Graphics, Locator, Window

Related to:

Format: SetCursor(Type)

Parameters:

Type

Required. Usually a numeric expression giving the cursor type to display according to the following table, but may also be a text argument giving the full path of the image file to be loaded as the cursor. Most of the common image files formats can be used. If giving the image file, the size of the cursor will be the system

Type	Cursor Type
-1	Set the cursor to the parent window's cursor
0	Normal Select (Standard arrow)
1	Busy (Waiting hourglass)
2	Text Select (Text I-beam)
3	Precision Select (Crosshairs)
4	Unavailable (Slashed circle)
5	Move (Four-pointed arrow pointing north, south, east, and west)
6	Diagonal Resize 2 (Double-pointed arrow pointing northeast and southwest)
7	Vertical Resize (Double-pointed arrow pointing north and south)
8	Diagonal Resize 1 (Double-pointed arrow pointing northwest and southeast)
9	Horizontal Resize (Double-pointed arrow pointing west and east)
10	Alternate Select (Vertical arrow)
11	Working in Background (Standard arrow and small hourglass)
12	Help Select (Standard arrow and small hourglass)

Comments: This statement will affect the cursor in the current window in which it is executed. The appearance of the cursor will depend on the current Windows™ cursor settings. The changed cursor will only be displayed within the client area of the window. Changing the cursor for a window will change the cursor for its child windows unless the child windows use SetCursor to change their cursor.

Example:

```
If editFlag EditText;
[
  SetCursor(2);
]
```

This is a script that might be executed prior to opening an editor window, for which a Text Select style cursor is desired.

SetDDEServer

Description: Sets the DDE topic name for a window.

Returns: Nothing

Usage: Script Only.

Function Groups: DDE

Related to: DDE | DDEPoke | DDEShareAdd | DDEShareDel

Format: SetDDEServer(Object, Title)

Parameters:

Object

Required. Any object expression that identifies the window.

Title

Required. Any text expression for the new topic name.

Comments: To ensure that proper DDE communication takes place, each window should have its own unique topic name. The

default topic name is the title of the window.

SetDefault

Description: Sets the default value for a variable.

Returns: Nothing

Usage: Script or steady state.

Function Groups: Compilation and On-Line Modifications, Variable

Related to: GetDefaultValue

Format: SetDefault(Variable, Default)

Parameters:

Variable

Required. Any expression for the variable value.

Default

Required. Any expression for the new default value.

SetDivert

(RPC Manager Library)

Description: Informs RPC Manager that the synchronization state of a service has been sampled during synchronization, and service RPCs for the specified client should be buffered until synchronization completes. Subroutine call only.

Returns: Nothing

Usage: Script Only.

Function Groups: Network

Related to: GetClientDiverts

Format: \RPCManager\SetDivert(Service, IP [, OptGUID]);

Parameters:

Service

Required. The name by which the service is known.

IP

Required. An IP by which the synchronizing client is known.

OptGUID

An optional GUID of the application in which the service instance is located. The default is the application to which the caller belongs.

Comments: This subroutine is a member of the RPC Manager's Library, and must therefore be prefaced by \RPCManager\, as shown in the "Format" section. If the application you are developing is a script application, the subroutine call must be prefaced by System\RPCManager\, and the System variable must be declared in AppRoot.src.
This subroutine is only called by the service instance that is server during synchronization.

SetEditMode

Description: Sets the graphics edit mode for a window.

Returns: Nothing

Usage: Script or steady state.

Function Groups: Editor, Window

Related to: AddEditorText | CurrentLine | Editor | ForceEvent | GoToOffset | MakeEditor

Format: SetEditMode(Object, Mode)

Parameters:

Object

Required. Any object expression that identifies the window.

Mode

Required. Any numeric expression for the new edit mode:

Mode	Edit Mode
0	Disable mouse actions
1	Enable mouse actions
2	Toggle mouse on/off

SetEnable

Note: Deprecated. Do not use in new code.

(Alarm Manager module)

Description: Tell the Alarm Manager to enable or disable an alarm. This function handles the attribution of changes to a user based on the tag's metadata. Use instead of the older Enable function when writing new code.

Returns: Invalid (see parameter, FeedBack)

Usage: Script Only.

Function Groups: Alarm

Related to: Register (Alarm Manager) | Enable

Format: \AlarmManager\SetEnable(TagName, AlarmObj, Disable, InhibitParm, FeedBack);

Parameters:

TagName

Required. The name of the tag that owns the alarm.

AlarmObj

Required. An instance of the alarm module.

Disable

Boolean. Set TRUE to disable the alarm, FALSE to enable.

FeedBack

Numeric. Indicates the result of the action. Set to 1 if the alarm was enabled or disable, and 0 if no operation was performed.

Comments: The SetEnable subroutine always returns Invalid.

SetFileAttribs

Description: Sets the attributes of the specified file.

Returns: Nothing

Usage: Script Only.

Function Groups: File I/O

Related to: GetFileAttribs

Format: SetFileAttribs(FileName, Attributes, Mode)

Parameters:

FileName

Required. Any text expression giving the name of the file. A known path alias for File-Related Functions may be provided in the form, :{KnownPathAlias}.

Attributes

Required. Any numeric expression which designates the attributes to be set. This parameter is formed by adding together numbers from the following table:

Value	Bit No	Attribute
0	-	Normal
1	0	Read only
2	1	Hidden
4	2	System
8	3	Archive

Optionally if mode is 1, may be a timestamp.

Mode

Controls the action of this function. If invalid or set to 0, the file attributes will be set as noted above. If set to 1, then this function will set the file's date.

Comments: Four file attributes may be set by this statement. All attributes will be changed at once.

Example:

```
If Watch(0, newFile);
[
  SetFileAttribs(Concat(MyPath, newFile), 9);
]
```

The above statement will cause file newFile to have its ReadOnly and Archive bit set every time its name changes.

SetHandle

Description Sets the position of graphics handles in a window.

Returns Nothing

Usage Script or steady state.

Function Groups Graphics

Related to:

Format: SetHandle(Object)

Parameters

Object

Required. Any object expression which identifies the window.

SetHelp

Description: Sets the help file name and (optionally) the context identifier for the window containing the specified object.

Returns: Nothing

Usage: Steady State only.

Function Groups: Help

Related to: Help

Format: SetHelp(Object, HelpFileName [, HelpContext])

Parameters:

Object

Required. The object value of a module instance that is running inside the window whose help context is to be affected.

HelpFileName

Required. The file name of the help file to use if the user presses F1 while the specified window is the active window. If Invalid, the parent window of the specified window will be checked for a help file reference. This continues recursively until the top of the window tree is reached.
If no help file name is found, the default VTScada help file is used. The default help reference can be set by adding the variable "WEBHelp" to the [System] section of the Setup.INI file, or by using the EnableHelp statement.
The help file name may be any .CHM or .HLP format help file, typically added to the VTScada installation folder. If you have created a NetHelp (DocToHelp format) or MadCapWebHelp (Flare format) help system, use the strings, "MyHelpFolder\NetHelp" or "MyHelpFolder\MadCapWebHelp" respectively, where MyHelpFolder points to the folder containing your custom help system.

To use the default VTScada Help system, enter simply: \DevHelpFile without quotation marks.

HelpContext

Optional help context. If absent or invalid, but the HelpFileName is valid, then the default home page of the help file or system is displayed when the user presses F1.
If valid and numeric, the help file is searched for a topic with a matching alias number and help is displayed for that topic. If there is no topic with a matching alias ID value, then NetHelp and MadCapWebHelp formats will open to the default home page. CHM formats will not open.
If valid and textual, and if the help file is either .CHM or .HLP format, then the help file is searched for an exact match on the text string in the topic index of the help file. If there is more than one text match, the index is positioned at the first partial string match. NetHelp and MadCapWebHelp formats will ignore a textual value for HelpContext and open to the default home page.
If the HelpFileName parameter is invalid, this parameter is ignored.

Comments: The setting of help file name and context for a window performed by this statement overrides those specified by Window statement parameters. When this statement stops, the settings for an affected window revert to the state [if any] set up by the Window statement parameters. If more than one of these statements references the same window, the last one to start wins the race.

The object is usually, Self(); if this statement used in the context of a page module.

SetINIProperty

(System Library)

Description: Given an INIFiles structure, this function sets the property with the specified name and section to the specified value. Does not affect the running system until WritePropertiesFile is called.

Returns: Reference to the INIProperty structures for the specified section.

Usage: Script Only.

Function Groups: Configuration Management, Variable

Related to: ReadPropertiesFile | WritePropertiesFile | GetINIProperty

Format: \System\SetIniProperty(pProperties, Section, Name, Value, Comment, pFail)

Parameters:

pProperties

Required. A pointer to an INIFile structure.

Section

Required. The name of the section in which the property belongs.

Name

Required. The name of the property being modified.

Value

Required. The value to set for the given property.

Comment

An optional text expression for the comment to write. If invalid, the existing comment (if any) for the property will be used.

pFail

A pointer to a Boolean value. Will be set TRUE if the operation fails.

Comments: This function does not change the property in configuration – it only changes the value in the INIFiles structure, which is obtained using a call to ReadPropertiesFile. The standard usage is to obtain the structure using ReadPropertiesFile, modify a value using SetINIProperty, then write the structure back to disk using WritePropertiesFile.

The INIFile structure returned is as follows:

```
INIFiles Struct [
  FileName    { full path and file name to the
settings file          };
  OEM         { TRUE if an OEM layer file
};
  Workstation { Name of the workstation or
invalid if global          };
  Layer       { Instance of application layer
owning the file         };
  Dynamic     { TRUE if a dynamic property
};
  Sections    { Dictionary of sections each ele-
ment of which
                is an array of Property struc-
tures        };
  Changed     { User sets to true if the file
has been changed,
                initialized to false
};
 ]
```

The INIProperty structure is…

```
INIProperty Struct [
    Name        { Variable name in the .star-
tup/.dynamic file  };
    Value       { Simple value
};
    Comment     { Text comment if present in the
file          };
    Hidden      { TRUE if not visible in Edit
Properties GUI   };
  ];
```

SetInstanceName

Description: Set the name of an instance of a module.

Returns: Nothing

Usage: Script or steady state.

Function Groups: Basic Module

Related to:

Format: SetInstanceName(Instance, Name)

Parameters:

Instance

Required. The object value for which the name is to be assigned.

Name

Required. The text string name of the instance.

Comments: Retained variables allow each separate instance of a module to retain its value on disk between VTScada executions or between instantiations of the module. Each instance must be assigned a name, and this SetInstanceName() function provides this facility.

SetInstanceRefBox

Description: Programmatically, set the module reference box of a single module instance.

Warning: This statement should be used by advanced users only since irrevocable alteration of your application may occur.

Returns: Invalid.

Usage: Script Only.

Function Groups: Compilation and On-Line Modifications, Graphics, Advanced Module

Related to: SetModuleRefBox

Format: SetInstanceRefBox(Module, Left, Bottom, Right, Top)

Parameters:

Module

Required. Any expression for the object value of the instance of the module.

Left

Required. Any numeric expression for the left side of the reference box for Module's graphics.

Bottom

Required. Any numeric expression for the bottom side of the reference box for Module's graphics.

Right

Required. Any numeric expression for the right side of the reference box for Module's graphics.

Top

Required. Any numeric expression for the top side of the reference box for Module's graphics.

Comments: This function adds a way to set the module reference box of a single module instance programmatically. If a module has SetInstanceRefBox called for an instance of it, then SetModuleRefBox will not affect that instance.

Comparing the two functions, SetModuleRefBox provides a way to programmatically set the default size of a module whereas SetInstanceRefBox provides a way to set the size of a specific instance of a module.

In any of the parameters are omitted, or if one or more is invalid, the instance-specific reference box will be cleared.

Examples:

The following is a widget that will draw the image, which is specified by its full path name in the text parameter, at its native size:

```
{================== Image Widget ======================}
(
  ImageChoice <:"Select Image":> Text;
)
[
  Title = "ImageWidget";
  Shared UserMethods (LIBRARIES);
  bWidth = 1 { default size is 1x1} ;
  bHeight = 1 ;
  bObj { bitmap object ref } ;
]
Init [
  If Watch(1) WidgetMain;
  [
    bObj = MakeBitmap(ImageChoice),
    bObj = MakeBitmap("\Bitmaps\Icons\Question icon.bmp"));}
    bObj = MakeBitmap(ImageChoice);
    bWidth = PickValid(BitmapInfo(bObj, 0), 1);
    bHeight = PickValid(BitmapInfo(bObj, 1), 1);
    SetInstanceRefBox(Self, 0, bHeight, bWidth, 0);
  ]
]
WidgetMain [

  GUIBitmap(0, 1, 1, 0 { Bounding box of image },
            1 - 0, bHeight, bWidth, 1 - 0, 1 { Scaling },
            0, 0 { No trajectory or rotation },
            1 { Bitmap is visible },
            0 { Reserved },
            4 { Left mouse button activates },
            0 { Focus ID number },
            0 { Focus trigger },
            bObj { Bitmap to show });
]
```

SetKeyParam

Description: The SetKeyParam function customizes various aspects of a session key's operations. The values set by this function are not persisted to memory and can only be used with in a single session. It is the VTScada analog of the CryptoAPI SetKeyParam call.

Returns: Nothing

Usage: Script Only.

Function Groups: Cryptography

Related to: DeriveKey | Decrypt | Encrypt | ExportKey | GenerateKey | GetCryptoProvider | GetKeyParam |

ImportKey

Format: SetKeyParam(Key, Param [, Value, Flags, Error])

Parameters:

Key

Required. The handle to the key for which values are to be set.

Param

Required. A parameter specifying the value to be set. Values are defined in WinCrypt.h

Value

An optional parameter containing the value to which the keys parameter is to be set. If omitted or invalid then the value 0 is used.

Flags

An optional parameter specifying the flags to be passed to CryptSetKeyParam. If omitted or invalid then the value 0 is used.

Error

Required. An optional variable in which the error code for the function is returned. It has the following meaning:

Error	Meaning
0	Key parameter successfully set.
1	Key or Param or Value parameters invalid.
X	Any other value is an error from CryptSetKeyParam.

Comments: The allowable values for Param vary with the key type.

Example:

```
[
  Constant KP_G = 12 { DSS/Diffie-Hellman G value };
]
Init [
  If 1 Main;
  [
    { Set the key parameter }
    SetKeyParam(Key2, KP_G, KeyG);
  ]
]
```

SetLibrary

Description: Sets the library for an application.

Warning: This statement should be used by advanced users only since irrevocable alteration of your application may occur.

Returns: Nothing

Usage: Script Only.

Function Groups: Compilation and On-Line Modifications

Related to:

Format: SetLibrary(Module, LibraryName)

Parameters:

Module

Required. Any expression that identifies the module or object for which the library is to be set.

LibraryName

Required. Any module expression that identifies the library.

Comments: This statement may only appear in a script.

SetModuleRefBox

Description: Sets the default reference box for a module.

Warning: This statement should be used by advanced users only since irrevocable alteration of your application may occur.

Returns: Nothing

Usage: Script Only.

Function Groups: Compilation and On-Line Modifications, Graphics, Advanced Module

Related to: GetModuleRefBox | GetXformRefBox | SetInstanceRefBox

Format: SetModuleRefBox(Module, Left, Bottom, Right, Top)

Parameters:

Module

Required. Any expression for the object value of the module.

Left

Required. Any numeric expression for the left side of the reference box for Module's graphics.

Bottom

Required. Any numeric expression for the bottom side of the reference box for Module's graphics.

Right

Required. Any numeric expression for the right side of the reference box for Module's graphics.

Top

Required. Any numeric expression for the top side of the reference box for Module's graphics.

Comments: A module's reference box defines an area that will exactly enclose all active graphics (graphics currently displayed) in the module before any rotations and trajectories have been applied. A module reference box, or MRB as it is sometimes called, is not a clipping region and objects can and often will extend outside of their MRB as a result of applied rotations or trajectories.

When a module is transformed, the transform is based on

the size of the module as determined by its reference box. If the module switches states, the active graphics will change, thereby changing the MRB for that module. The result is that the transform will change such that graphic objects will grow or shrink, so that the module's reference box will always exactly fill the reference box of the transform. In the case of graphics that have had a rotation or trajectory applied to them, the graphics will be transformed correctly, but the MRB may no longer contain the objects in their modified positions.

This statement allows the user to set the reference box of a module to a constant size, so that as graphics become active and inactive, the transform will not cause the graphic objects to grow and shrink - since the transform is based on the module's reference box, and this is now fixed, the transform will be similarly fixed.

This call should be followed by a call to SaveModule, otherwise the module reference box change will only be written when the application containing this statement is stopped (in the case of base VTScada code this means when VTScada is shut down).
Great care should be exercised when using this statement, since the module reference box for the module itself is altered, not just the instance in which the statement is called. All instances of this module will have their reference box permanently changed.

Example:

```
Init [
  If 1 Main;
  [
    SetModuleRefBox(Self(), 0, 599, 799, 0);
  ]
]
```

This sets the current module's reference box to a fixed size equal to the default window size.

SetModuleText

Description: Sets the module's .SRC file information.

Warning: This statement should be used by advanced users only since irrevocable alteration of your application may occur.

Returns: Nothing

Usage: Script Only.

Function Groups: Compilation and On-Line Modifications, Advanced Module

Related to: AdjustCode | GetModuleText | GetOneParmText | GetParmText | GetStateText | GetTransitText | GetVariableText | SetOneParmText | SetParmText | SetStateText | SetTransitText | SetVariableText | TextOffset | TextSize

Format: SetModuleText(Module, Mode, Value)

Parameters:

Module

Required. Any expression for the module or object value.

Mode

Required. Any numeric expression for the value to set

Mode	Value to set
0	.SRC file name Mode 0 cannot be used to make a module have a .SRC file name in a different directory than where the module presently resides.
1	Character offset to beginning of module definition
2	Size of module definition in characters
3	Character offset to first parameter declaration
4	Character offset to first variable declaration
5	Character offset to first state
6	Character offset to first child module definition
7	Character offset to beginning of variable declaration block
8	Size of variable declaration block in characters
9	Character offset to beginning of parameter declaration block
10	Size of parameter declaration block in characters
11	Full path and file name of .SRC file

Value

Required. Any expression for the value the module information specified by Mode will be set to.

SetOneParmText

Description: Sets the text for one parameter of a function.

Warning: This statement should be used by advanced users only since irrevocable alteration of your application may occur.

Returns: Nothing

Usage: Script Only.

Function Groups: Compilation and On-Line Modifications, Advanced Module

Related to: AdjustCode | GetModuleText | GetOneParmText | GetParmText | GetStateText | GetTransitText | GetVariableText | SetModuleText | SetParmText | SetStateText | SetTransitText | SetVariableText | TextOffset | TextSize

Format: SetOneParmText(Code, Index, Text)

Parameters:

Code

Required. Any expression for the code value of the statement.

Index

Required. Any numeric expression for the parameter number to change, beginning with 0.

Text

Required. Any text expression. The text in the .SRC file for the parameter indicated by Code and Index will be replaced by Text.

SetOPCData

Description: Sets an item value in the VTScada OPC Server.

Warning: For use by advanced programmers only

Returns: Boolean

Usage: Script Only.

Function Groups: Network

Related to: OPCServer

Format: SetOPCData(BranchHandle, ItemName, Value, Quality, Timestamp)

Parameters:

BranchHandle

Required. A handle returned from an OPCServer call.

ItemName

Required. The internal name for the OPC item being set. It does not necessarily correspond to the OPC item ID. It does correspond to what the OPCGetIn-ternalName callback module returns for a given OPC item ID.

Value

Required. The new value of the item (numeric or text).

Quality

Required. The quality of the value. Should be one of the following:

Quality	Meaning
0x00	Bad
0x04	Bad - Configuration Error (The item has been deleted)
0x40	Uncertain - Questionable quality
0xD8	Good but local override
0xC0	Good
0xC3	Good but constant value

Timestamp

Required. The UTC timestamp corresponding to the

value. Will default to the current time if Invalid.

Comments: Returns TRUE if the item being updated is currently included in an OPC client group, or FALSE if not.

Example:

```
SetOPCData(Handle    { Returned from an OPCServer call },
           "myitem1",
           23.1,
           0xC0,
           CurrentTime() + TimeZone(0));
```

This example updates the value of an OPC item with the internal name "myitem1" to be 23.1, with good quality and the current time as the timestamp.

SetOverride

Description: Allows the overriding of OpCodes with a specified script module within a static module tree.

Warning: This statement should be used by advanced users only. Effective use of this function requires a thorough understanding of VTScada programming.

Returns: Nothing

Usage: Script Only.

Function Groups: Advanced Module

Related to: GetOverrides

Format: SetOverride(TargetModule, OpCode, [Override, Recursive])

Parameters:

TargetModule

Required. Any expression that can be resolved to the module value that will be modified.

OpCode

Required. Any numeric value that represents the built-in function to replace.

Override

The module value that will be called in place of OpCode. If invalid or missing, any existing overrides of OpCode in TargetModule will be removed.

Recursive

Required. If true, the OpCode will be replaced with calls to the override function in all of the child modules of the target module. Otherwise, only the target module will be altered. Defaults to true.

Comments: Adds the ability to override a built-in function inside a module with a call to a different module. The purpose of this feature is to help with testing of modules that use time, streams, etc. so that these functions can be overridden with more controllable inputs.

SetParameter

Description: Sets a parameter in a statement.

Warning: This statement should be used by advanced users only since irrevocable alteration of your application may occur.

Returns: Nothing

Usage: Script Only.

Function Groups: Compilation and On-Line Modifications, Advanced Module

Related to:

Format: SetParameter(Statement, ParmNum, ParmType, Value)

Parameters:

Statement

Required. Any expression for the statement value. May be a code value or a function value.

ParmNum

Required. Any numeric expression for the parameter

number to change, beginning with 0.

ParmType

Required. Any numeric expression for the VTScada Value Types – Numeric Reference of the new parameter.

Value

Required. Any expression for the new parameter's value.

Comments: This statement is used to modify the code for a function.

SetParmText

Description: Sets the text for the parameters of a function.

Warning: This statement should be used by advanced users only since irrevocable alteration of your application may occur.

Returns: Nothing

Usage: Script Only.

Function Groups: Compilation and On-Line Modifications, Advanced Module

Related to: AdjustCode | GetModuleText | GetOneParmText | GetParmText | GetStateText | GetTransitText | GetVariableText | SetModuleText | SetOneParmText | SetStateText | SetTransitText | SetVariableText | TextOffset | TextSize

Format: SetParmText(Code, Text)

Parameters:

Code

Required. Any expression for the code value of the statement.

Text

Required. Any text expression. The text in the .SRC file for the parameters of Code will be replaced by Text.

Comments: None

SetParserParm

Description: Sets the value for the last parameter on the parser stack and returns its own error code.

Warning: This statement should be used by advanced users only since irrevocable alteration of your application may occur.

Returns: Numeric error code

Usage: Script Only.

Function Groups: Compilation and On-Line Modifications

Related to: Compile

Format: SetParserParm(ParserStack, OpCode, Value, Offset)

Parameters:

ParserStack

Required. Any expression for the parser stack value returned by the compiler.

OpCode

Required. Any numeric expression for the type of parameter to set as given by the following table

Opcode	Parameter Type
0	Integer
1	Double
2	Text
3	Variable

Value

Required. Any expression. It will be evaluated as the type specified by Opcode.

Offset

Required. Any numeric expression for the number of characters read from the stream at this point (i.e. the value returned in the last parameter of the Compile function).

Comments: The return value is the error code for this function or 0 if no error.

SetRefRect

Description: Sets the first four constant parameters of a layered graphic statement.

Warning: This statement should be used by advanced users only since irrevocable alteration of your application may occur.

Returns: Nothing

Usage: Script or steady state.

Function Groups: Compilation and On-Line Modifications, Graphics

Related to:

Format: SetRefRect(CodePointer, Left, Bottom, Right, Top)

Parameters:

CodePointer

Required. Any expression for the code pointer value which identifies the graphics statement.

Left

Required. Any numeric expression for the left side coordinate.

Bottom

Required. Any numeric expression for the bottom side coordinate.

Right

Required. Any numeric expression for the right side coordinate.

Top

Required. Any numeric expression for the top side coordinate.

Comments: This statement affects the .SRC file as well.

SetRemoteValue

(RPC Manager Library)

Description: This subroutine sets the specified variable within an application instance on a workstation to the specified value. Subroutine call only.

Returns: Nothing

Usage: Script Only.

Function Groups: Network

Related to: ConnectToMachine | DisconnectFromMachine | GetServer | GetServersListed | GetStatus | IsClient | IsPotentialServer | IsPrimaryServer | Register (RPC Manager) | Send

Format: \RPCManager\SetRemoteValue(VariableScope, Variable, Value, Name [, OptGUID])

Parameters:

VariableScope

Required. The scope in which to find the variable.

Variable

Required. The name of the variable.

Value

Required. The value to set for the variable. Subject to the restriction of the type of values that can be set via RPC.

Name

Required. Any of the names or IPs by which the work-

station is known to the RPC Manager.

OptGUID

Any optional parameter that provides the GUID of the application in which the variable is to be set. The default is the application to which the caller belongs.

Comments: This subroutine is a member of the RPC Manager's Library, and must therefore be prefaced by \RPCManager\, as shown in the "Format" section. If the application you are developing is a script application, the subroutine call must be prefaced by System\RPCManager\, and the System variable must be declared in AppRoot.src.
If the 16-byte binary format of the GUID is not known, the GetGUID function may be used to obtain it.

Example:

```
If 1 Main;
[
  \RPCManager\SetRemoteValue("\" { root of app context },
  "DispMgrFullScreen" { var name },
  1 { value }, "TestMachine");
]
```

This will cause the variable (flag) DispMgrFullScreen, which is found in the root scope of the application, to be set to 1 on the machine called TestMachine.

Related Information:

Refer also to "RPC Manager Service" for a listing of Service Control Methods, RPC Methods, and Deprecated RPC Methods.

SetReturnValue

Description: Sets the return value of a specified object if the object is not currently running a Return() statement in steady state.

Returns: Nothing

Usage: Script or steady state.

Function Groups: Basic Module

Related to: HasReturnStatement

Format: SetReturnValue(Object, Value)

Parameters:

Object

Required. A reference to the object whose return value is to be changed.

Value

Required. The new return value.

Comments: Returns 1 if successful, 0 on failure, or Invalid if the first parameter cannot be resolved to an object.

SetShelved

(Alarm Manager module)

Description: AlarmManager plug-in, that handles the shelving and unshelving of alarms.

Returns: Nothing

Usage: Script Only.

Function Groups: Alarm Manager

Related to: IsShelved

Format: \AlarmManager\SetShelved(AlarmName, Shelve[, ExpiryTime, Timestamp, AccountID, Device, MachineID]);

Parameters:

AlarmName

Required text. The name of the alarm to be shelved.

Shelve

Required. Boolean that indicates whether to shelve (TRUE) or unshelve (FALSE) the alarm.

ExpiryTime

Optional. The UTC timestamp for when the alarm should automatically unshelve.

Timestamp

Optional. The UTC timestamp of this event. Defaults to now.

AccountID

Optional. ID of the user who initiated this event. Defaults to the user currently logged on at the work-station.

Device

Optional. The name of the client device calling this function.

MachineID

Optional. The machine ID of the workstation calling this function.

Comments:

Examples:

```
Code\AlarmManager\SetShelved(AlarmName, TRUE, CurrentTime(1) + 300);
```

SetStateText

Description	Sets the information about the text of a state in a .SRC file.
Warning	This statement should be used by advanced users only since irrevocable alteration of your application may occur.
Returns	Nothing
Usage	Script or steady state.
Function Groups	Compilation and On-Line Modifications, States
Related to:	AdjustCode | GetModuleText | GetOneParmText | GetParmText | GetStateText | GetTransitText |

GetVariableText | SetModuleText | SetOneParmText | SetParmText | SetTransitText | SetVariableText | TextOffset | TextSize

Format: SetStateText(State, Mode, Value)

Parameters

State

Required. Any expression for the code value of the state.

Mode

Required. Any numeric expression for the parameter to set in the state

Mode	Parameter to set
0	Character offset to beginning of state
1	Size of state text in characters
2	Character offset to first statement text

Value

Required. Any numeric expression giving the new value (as determined by Mode) for the state.

Comments:
None

SetSyncComplete

(RPC Manager Library)

Description: Informs RPC Manager that service synchronization is complete as far as the local service instance is concerned. Subroutine call only.

Returns: Nothing

Usage: Script or steady state.

Function Groups: Network

Related to:

Format: \RPCManager\SetSyncComplete(Service [, OptGUID, Value]);

Parameters:

Service

Required. A name by which the service is known.

OptGUID

An optional parameter indicating the GUID of the application in which the service instance is located. The default is the application to which the caller belongs.

Value

An optional Boolean that will indicate the completion state (1 default).

Comments: This subroutine is a member of the RPC Manager's Library, and must therefore be prefaced by \RPCManager\, as shown in the "Format" section. If the application you are developing is a script application, the subroutine call must be prefaced by System\RPCManager\, and the System variable must be declared in AppRoot.src.

Related Information:

See: "Client Changes" in the VTScada Programmer's Guide.

SetTransfer

Description: Sets the destination for an action.

Warning: This statement should be used by advanced users only since irrevocable alteration of your application may occur.

Returns: Nothing

Usage: Script or steady state.

Function Groups: Compilation and On-Line Modifications, States

Related to:

Format: SetTransfer(Action, Destination)

Parameters:

Action

Required. Any expression for the code value of the action.

Destination

Required. Any expression for the code value of the destination state.

Comments: none

SetTransitText

Description: Sets the information about the document file definition of an action (predicate).

Warning: This statement should be used by advanced users only since irrevocable alteration of your application may occur.

Returns: Nothing

Usage: Script Only.

Function Groups: Compilation and On-Line Modifications

Related to: AdjustCode | GetModuleText | GetOneParmText | GetParmText | GetStateText | GetTransitText | GetVariableText | SetModuleText | SetOneParmText | SetParmText | SetStateText | SetVariableText | TextOffset | TextSize

Format: SetTransitText(Action, Mode, Value)

Parameters:

Action

Required. Any expression for the code value of the action. This corresponds to value type VTScada Value Types – Numeric Reference.

Mode

Required. Any numeric expression for the parameter to set in the action:

Mode	Parameter
0	Script size in characters
1	Character offset of first script statement
2	Trigger size in characters
3	Character offset to trigger
4	Destination size in characters
5	Character offset to destination

Value

Required. Any numeric expression giving the new value for the parameter.

Comments: This statement may only appear in a script.

SetVariableClass

Description: Sets the class number of a variable and returns its previous class number.

Returns: Numeric

Usage: Script Only.

Function Groups: Compilation and On-Line Modifications, Variable

Related to: VariableClass

Format: SetVariableClass(Variable, Class)

Parameters:

Variable

Required. Any expression for the Variable to set.

Class

Required. Any numeric expression for the class. It must be in the range of 0 to 65535.

Comments: The class number for a variable defaults to 0.

Example:

```
If ! classSetFlag;
[
  SetVariableClass(FindVariable("newVar", Self(), 0, 1), 20);
  classSetFlag = 1;
]
```

This sets the class of variable newVar to 20.

SetVariableText

Description: Sets information about the document file definition of a module.

Warning: This statement should be used by advanced users only since irrevocable alteration of your application may occur.

Returns: Nothing

Usage: Script Only.

Function Groups: Compilation and On-Line Modifications, Variable

Related to: AdjustCode | GetModuleText | GetOneParmText | GetParmText | GetStateText | GetTransitText | GetVariableText | GetTransitText | GetVariableText | SetModuleText | SetOneParmText | SetParmText | SetStateText | SetTransitText | TextOffset | TextSize

Format: SetVariableText(Variable, Mode, Value)

Parameters:

Variable

Required. Any expression for the Variable value.

Mode

Required. Any numeric expression for the variable to set:

Mode	Expression Variable
0	Character offset to variable declaration
1	Variable declaration size in characters

Value

Required. Any numeric expression giving the new value for the variable.

SetVariableType

Description Sets the data type for the variable, so that only values of that data type can be stored in the variable.

Warning This statement should be used by advanced users only.

Returns Nothing

Usage Script Only.

Function Groups Compilation and On-Line Modifications, Variable

Related to: GetVariableType | CrossReference

Format: SetVariableType(Variable, Value)

Parameters

Variable

Required. A Variable handle, such as would be returned from the FindVariable or AddVariable functions.

Value

Required. The type that values put into this variable should be cast to.

Comments Casts the variable's value to the given type. If the cast cannot be performed, the variable's type will be set to Invalid.

The implementation of this function is very similar to SetVarMetadata.

SetVarMetadata

Description: Every variable object contains an embedded value. This function is used to set those values.

Warning: This statement should be used by advanced users only.

Returns: Nothing

Usage: Script Only.

Function Groups: Dictionary

Related to: GetVarMetadata | FindVariable | AddVariable

Format: SetVarMetadata(Variable, Value)

Parameters:

Variable

Required. A Variable handle, such as would be returned from the FindVariable or AddVariable functions.

Value

Required. Any value to store within the variable's metadata.

Comments: Commonly used in conjunction with SetVarMetadata, FindVariable or AddVariable. Note that type data for each variable is stored within the variable using metadata.

Example:

```
<
TestMod
[
  X;
  Y;
  Var;
]
Main [
  If ! Valid(X);
```

```
  [
    X = "This is the value of X"
    Var = FindVariable("X", Self(), 0, 0);
    SetVarMetadata(Var, "This is the metadata in variable X");
    Y = GetVarMetadata(Var);
  ]
  ZText(50, 100, Concat("X: ", X), 14, 0);
  ZText(50, 120, Concat("Y: ", Y), 14, 0);
]
>
```

SetVicParms

Description: Sets parameters for the VTScada Internet Client.

Returns: Nothing

Usage: Script Only.

Function Groups: VTScada Internet Client

Related to:

Format: SetVicParms(ReadTimeout, SessionTimeout, Mode[, AllowedOrigins])

Parameters:

ReadTimeout

Required. This is the length of time in seconds that the client will wait for the server to respond. Has a minimum of 5 seconds and a default of 15.

SessionTimeout

Required. This is the length of time in seconds that the server will wait for the client to respond. Has a minimum of 10 and a default of 60.

Mode

Required. VIC connection mode values are as follows (Defaults to 0)

Mode	Meaning
0	legacy
1	priority
2	sticky

AllowedOrigins

Optional. An array of the names of possible hosts that Anywhere clients will connect from. The VIC server will not communicate with Anywhere clients running from hosts not in that array.

Comments: This module returns 1 if the parameters were set or Invalid if not.
Note that the changes made by this statement do not affect current VTScada Internet Connections. Only new connections will use the changed parameters. This statement is used internally by the VTScada System layer. Any changes made by this statement will be overwritten by the VTScada System layer on startup of the VTScada System layer and upon change of any setting in the Internet Client dialog.

Setting either the ReadTimeout or the SessionTimeout to values outside the recommended guidelines can have unpredictable effects on the ability of clients to maintain a reliable communication session with the server.
For the Anywhere Client, all entries in the server list are automatically added to the array of AllowedOrigins. Additional hosts may be added by specifying

HostName = 1 in the [Clients-AdditionalAllowedOrigins] section of Setup.INI.

SetWSDL

(System Library)

Description: Connects a Realm with a WSDL file and a set of VTScada modules in order to enable a web service interface.

Returns Numeric (0 for success, 1 for failure)

Usage: Script Only.

Function Groups: XML

Related to: RemWSDL | XMLProcessor | XMLAddSchema | XMLParse | XMLWrite

Format: \System\Webservice\SetWSDL(WSDLFilePath, Realm, CallScope, Service[, pResponse, XSDFileName, WSDrvrVarName])

Parameters:

WSDL File Path

Required. This is a file path that indicates where the WSDL file to be used for this service is located. The file path must be encoded in URL format. Only one WSDL file can be declared and it must be passed in the first parameter.

Realm

Required. The name of the VTScada Realm which will be used to expose the web service. Only one Realm can be specified and it must be passed as the second parameter.

Call Scope

Required. An instance (object) of the module in which all of the modules to be called are nested. This provides the scope for the remote module calls by the

WebService engine. Only one call scope object can be specified and it must be passed as the third parameter.

Service

Required. The name of the web service to be presented as portrayed in the WSDL's Service tag. This is a string that must match the "name" attribute of the representative Service tag and tells the SetWSDL function where to start searching for data.

pResponse

Optional. A pointer to a variable that will be loaded with the error description should SetWSDL fail. This description takes the form of a single human readable string. The variable will be set to invalid if no error occurs.

XSDFileName

Optional text. File name for the output XSD file. If not specified, a temporary file is used.

WSDrvrVarName

Optional text. Name of variable to be added into the call scope. Useful if setting multiple WSDL/Realms into same scope.

Comments: Linkage is first applied between the WSDL file and the VTScada modules by generating an XML schema using the WSDL and the parameters provided to this function. The Realm's address is then registered with the VTScada HTTP server to connect the whole thing to the network.
Inclusion of the parameter, pResponse is recommended, for the sake of obtaining error messages when debugging.

Example:

```
Init [
  If Watch(1);
  [
    ServiceActive = \System\WebService\SetWSDL(
       "file://C:/vts/StationExample/TagQueryServices.wsdl",
       "QueryServicesRealm",
       Self,
       "TagQueryServices",
       &ErrMsg);
  ]
]
```

SetXLoc

Description: Sets the X screen location of the locator (mouse).

Returns: Nothing

Usage: Script or steady state.

Function Groups: Graphics, Locator

Related to: SetYLoc | XLoc | YLoc

Format: SetXLoc(X)

Parameters:

X

Required. Any numeric expression giving the new X coordinate of the locator (mouse) on the screen.

Comments: This statement has no effect if the locator is not installed.

Example:

```
If XLoc() < 400;
[
  SetXLoc(400);
]
```

This action keeps the cursor in the right 400 pixels of the screen.

SetYLoc

Description: Sets the Y screen location of the locator (mouse).

Returns: Nothing

Usage: Script or steady state.

Function Groups: Graphics, Locator

Related to: SetXLoc | XLoc | YLoc

Format: SetYLoc(Y)

Parameters:

Y

Required. Any numeric expression giving the new Y coordinate of the locator (mouse) on the screen.

Comments: This statement has no effect if the locator is not installed.

Example:

```
If YLoc() > 300;
[
  SetYLoc(300);
]
```

This action keeps the cursor in the top 300 pixels of the screen.

ShiftStream

Description: Inserts or deletes characters from a stream and returns its own error code.

Returns: Nothing

Usage: Script or steady state.

Function Groups: Stream and Socket

Related to: Seek | StreamEnd

Format: ShiftStream(Stream, Pos, Offset)

Parameters:

Stream

Required. Any expression giving the stream to shift.

Pos

Required. Any numeric expression giving the stream position to seek to before beginning the shift.

Offset

Required. Any numeric expression giving the number of characters to insert. If this is negative, characters will be deleted.

Comments: The returned value is true if successful, false if not.

ShowLexicon

(VoiceTalk Module)

Description: Displays a SAPI text-to-speech engine lexicon dialog to permit modification of pronunciation. This function will return immediately, and the lexicon window will be managed in its own thread, preventing the calling thread from being blocked.

Returns: Nothing

Usage: Script Only.

Function Groups: Speech and Sound

Related to: Configure | GetDevices | GetVoices | Reset | Speak | VoiceTalk

Format: VoiceTalkStream\ShowLexicon([Title])

Parameters:

VoiceTalkStream

Required. A speech stream returned from VoiceTalk.

Title

An optional parameter that is any text expression to be displayed in the title bar of the dialog box.

Comments: This function spawns a new thread to show the lexicon, so it will not block other statements from executing.
The pronunciations are stored in a file known as a "Lex-

icon". This file is stored in the VTScada directory. It is therefore available to all users and all VTScada applications. Changes made will affect all instances of VTScada applications that use speech.
Each time this window is opened, the lexicon is refreshed from disk. When the dialog is closed, the lexicon is written back to disk.
On any single installation of VTScada, only one lexicon dialog will be displayed at any time, regardless of how many applications support speech.
Any word may be entered into the lexicon with a custom pronunciation. A word is any set of letters with no whitespace (i.e. no spaces or tabs) within it. Words are case-sensitive. For example, if the word "VTS" is entered with the phonetic spelling "V T S", the way the word "vts" will sound will not be affected.

Example:

```
sHandle = \VoiceTalk();
If Valid(sHandle) && ZButton(10, 40, 110, 10, "Pronounce", 1);
[
  sHandle\ShowLexicon("Modify Pronunciation");
]
```

When the button is pressed, the lexicon dialog for the text-to-speech engine open on the SAPI text-to-speech stream will be displayed.

ShowPage

Description: When called with a Page Name, will cause that page to be displayed in the caller's display session context.

Returns: Object reference

Usage: Script Only.

Function Groups: Graphics

Related to:

Format: \Code\DisplayManager\ShowPage(PageName[, ForceWin,

Parm0, ... Parm99]);

Parameters:

PageName

Required. Any text expression for the page to show.

ForceWin

Optional. Any Boolean value, which if valid and true, requests a new window for the page.

Parm0 through Parm99

Optional. Any parameters to be passed to the page as it is opened.

Comments: The page's window flags will have priority over the value of ForceWin. A page configured to never open in a pop-up won't, even though you set ForceWin to TRUE.
Returns the window object that calls the page.

Examples:

```
If IconPressed[#AlarmIcon];
[
  { Alarm icon -- switch to alarm page if clicked }
  \DisplayManager\ShowPage("AlarmPage");
]
```

SilenceSound

(Alarm Manager module)

Description This subroutine will silence the current sounding alarm.

Returns Numeric

Usage Script Only.

Function Groups Alarm

Related to:

Format: \AlarmManager\SilenceSound();

Parameters None

Comments The SilenceSound subroutine always returns "1".
The application property, AlmSilenceAllow, must be set to a numeric value for the mute setting to be communicated to all computers on the network. Any workstation will set the silence flag that has an AlmSilenceAllow value greater than or equal to the value set on the workstation executing the code.

SimpleOpChange

Description: Immediately deploys a single parameter change on a single tag without disturbing any other tag

Returns: Object

Usage: Script Only.

Function Groups: Configuration

Related to: OpChange

Format: SimpleOpChange(TagName, NewValue, ParameterName)

Parameters:

TagName

Required. The full name of the tag, in which the parameter will be changed.

NewValue

Required. The new value for the parameter.

ParameterName

Required. The name of the parameter that is to be changed.

Comments: While a direct call to OpChange is more efficient (it allows, for example, more than one parameter to be changed in the same operation), this function has a parameter set more like the older calls. The object value being returned

will go invalid when the asynchronous operational change is complete.
This function is declared in the VTS Library layer.

SimulateMouse

Description: Sets the pointer location and then sends a button press with modifiers such as Ctrl, Shift or Alt.

Returns:

Usage: Script Only.

Function Groups: Graphics,

Related to: SetXLoc| SetYLoc

Format: SimulateMouse(Root, mouseX, mouseY, MouseButtons, ModifierKeys)

Parameters:

Root

Required. The window object within which the mouse should point.

mouseX

Required. Any numeric expression for the horizontal location of the mouse pointer

mouseY

Required. Any numeric expression for the vertical location of the mouse pointer.

MouseButtons

Required. Any numeric expression for the mouse button(s) to be simulated.

Bit	Value	Mouse Button
--	0	no buttons
2^0	1	left down

2^1	2	right down
2^2	4	middle down
2^6	64	release any buttons, as specified above, in the same call to SimulateMouse()
2^7	128	delete key

ModifierKeys

Required. Any numeric expression for the keyboard modifier to be applied to the mouse click. See comments.

Bit	Value	Modifier Key
--	0	no key
2^0	1	shift
2^1	2	control
2^2	4	Alt
2^3	8	Left
2^4	16	Right
2^5	32	Up
2^6	64	Down

Comments: Similar to SetXLoc and SetYLoc, but all in one, plus mouse buttons and modifier keys. Used primarily for testing.

Only a single modifier can be specified in any call. The key is not released automatically, but rather is release when called again with bit 15 (0x8000) set. For example, SimulateMouse(x, x, 1) sets shift and SimulateMouse(x, x, 0x8001) releases shift.

Examples:

```
...
mouseX = 100;
mouseY = 150;
DblClick(mouseX, mouseY);
```

```
...
<
{======================= DblClick ============================}
{ Simulates a double click of the left mouse button }
{=============================================================}
DblClick
(
mouseX { Left Mouse Coord };
mouseY { Top Mouse Coord };
)
[
  Done = FALSE;
  Phase = 0;
  Constant #Btn = 0x01; { Left click }
  Constant #Btn_Up = 0x40; { release button }
  Constant #Key = 0x00; { No keyboard modifier }
]
First [
  If Timeout(!Done, 0.1);
  [
    Case(Phase,
      { Case 0 } Execute(
        { Does first mouse click }
        SimulateMouse(Root, mouseX, mouseY, #Btn, #No_Keys);
        SimulateMouse(Root, mouseX, mouseY, #Btn + #Btn_Up, #No_
Keys);
      ),
      { Case 1 } Execute(
        { does second mouse click }
        SimulateMouse(Root, mouseX, mouseY, #Btn, #No_Keys);
        SimulateMouse(Root, mouseX, mouseY, #Btn + #Btn_Up, #No_
Keys);
      ),
      { Case 2 }
        Result = TRUE;
    );
    Phase++;
  ]
]
>
```

Sin

Description: Returns the trigonometric sine of an angle in radians.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Trigonometric Math

Related to: ACos | ASin | ATan | Cos | Tan

Format: Sin(Angle)

Parameters:

Angle

Required. Any numeric expression giving the angle in radians.

Comments: The returned value is a number in the range of -1.00 to +1.00. To convert an angle from degrees to radians multiply by \pi / 180 or (approximately) 0.0174533.

Example:

```
x = Sin(270 * \pi / 180);
```

The value of x will be - 1.

SizeWindow

Description: Changes the visible size of a window on the screen.

Returns: Nothing

Usage: Script or steady state.

Function Groups: Window

Related to: MoveWindow | WindowOptions

Format: SizeWindow(Win, Width, Height[, VirtualWidth, VirtualHeight])

Parameters:

Win

Required. Any expression which gives an object value contained in the window to size.

Width

Required. Any numeric expression that gives the new width of the window.

Height

Required. Any numeric expression that gives the new height of the window.

VirtualWidth

Any numeric expression that gives the width inside the new window in user coordinates (which may be pixels). If VirtualWidth is larger than the client area specified, a horizontal scroll bar appears.

VirtualHeight

Any numeric expression that gives the height inside the new window in user coordinates (which may be pixels). If VirtualHeight is larger than the client area specified, a vertical scroll bar appears.

Comments: This statement will change the size of the window to the size given. Any objects drawn in the window are not scaled - this resizing simply changes the area displayed. Vertical and horizontal scroll bars will be added to the window's borders so that the entire area can be viewed.
If either VirtualWidth or VirtualHeight are specified, then both must be provided. Attempting to set one without the other will cause the one you set to be ignored. If these parameters are omitted, the virtual size of the window will be changed to whatever value is in the Width and Height parameters.

Example:

```
If ZButton(25, 50, 145, 100, "Resize", 1);
[
  SizeWindow(Self(), 300, 500);
]
```

This will display a button in the upper left corner of the window that when pressed, resizes the window to be 300 pixels wide by 500 pixels high.

Slay

Description: Stops a launched module, and possibly any parent modules.

Returns: Nothing

Usage: Script Only.

Function Groups: Basic Module

Related to: Stop | WindowClose

Format: Slay([Object, KillParents])

Parameters:

Object

An optional parameter which is any object expression for the launched module. The default value of Object is Self().

KillParents

An optional logical expression for the action to take. If true, Slay will attempt to stop parents as well. If any parent libraries are not launched, the Slay statement won't search for further modules instances. The default value for KillParents is 0.

Comments: When this statement is encountered in a script, the module is immediately stopped. Slay and Return are the only statements that will cause a script to terminate mid-way through its execution. Others like ForceState will not. Slay() is the same as Slay(Self(), 0).

Example:

```
If ZButton(20, 50, 120, 70, "Motor On/Off", 1);
[
  IfElse(!motorOn, Execute(
         motorOn = 1,
        motorPtr = RunMotor() { Module launched implicitly }),
  { else } Execute(
        motorOn = 0,
         Slay(motorPtr, 0)));
]
```

When the button is selected, the statement in the script will check if the motor is running or not, and if it isn't, will launch the module that starts

it. Notice that the module is launched implicitly because it is called from inside of a script. If that module is already running it will stopped by the Slay .

SocketAttribs

Description: Returns information about a TCP/IP socket's attributes.

Returns: Text or Buffer (see comments)

Usage: Script Only.

Function Groups: Stream and Socket

Related to: ClientSocket | CloseStream | ServerSocket | SocketAttribs | SocketServerEnd | SocketServerStart | SocketWait | SRead | StreamEnd | SWrite | TCPIPReset

Format: SocketAttribs(Stream, Option)

Parameters:

Stream

Required. Any stream expression for the socket. If this isn't a socket stream, invalid is returned.

Option

Required. Any numeric expression for the desired attribute:

Option	Attribute
0	Remote (text) workstation name
1	Remote port number
2	Remote workstation IP
3	Local machine name
4	Local machine IP
5	Number of bytes in the output buffer
6	Remote (text) workstation name (buffered)
7	Remote workstation IP (buffered)
8	Local machine name (buffered)
9	Local machine IP (buffered)
10	Remote machine Name (for incoming UDP datagrams)
11	Remote Port (for incoming UDP datagrams)
12	Remote IP (for incoming UDP datagrams)
13	Activate optional filters specified in ClientSocket()
14	UDP local port number
15	Socket type (0 TCP, 1 UDP)

Comments: The return value for options 0 and 6 is the machine name stored as a text value. If the name cannot be found it will return the internet address of the socket as a text string.

Options 6 through 9 differ from options 0 and 2 through 4 only in the fact that they are buffered. This means that an initial inquiry will be made to the operating system and stored in a buffer, and after that, all inquiries will be handed the value stored in the buffer. This makes options 6 through 9 significantly faster than the other options, however, any online changes to the workstation's attributes will be not be discovered by options 6 to 9, only 0 and 2 to 4 would return the new values.
If using option 13 to activate optional filters, note that (as of VTScada version 11) TLS/SSL is the only one supported.

SocketPingSetup

Description: Starts the transmission of automatic keep-alive "ping" messages through a socket stream.

Returns: Nothing

Usage: Script Only.

Function Groups: Stream and Socket

Related to: BuffStream | ServerSocket | ClientSocket

Format: SocketPingSetup(SocketStream, PingStream, TimeInterval)

Parameters:

SocketStream

Required. Any valid socket stream, typically obtained from a ClientSocket or ServerSocket.

PingStream

Required. A stream which contains the "ping" packet to be transmitted.

TimeInterval

Required. The interval, in seconds, between transmissions.

Comments: SocketPingSetup is used to enable the regular transmission of a small packet (a "ping") down a socket stream. The ping is only transmitted if there has been no other transmission for the specified time interval. Pinging continues automatically until the socket stream is closed. This function is useful for keeping a connection open which may be closed by the computer operating system due to inactivity.

Examples:

```
Init [
  If 1 Wait;
  [
    Socket = ClientSocket(0, TargetMachine, Port, TransmitLen,
ReceiveLen, 0);
  ]
]
Wait [
  If PickValid(ValueType(Socket == 8 {stream}, 0) Open;
  [
    SocketPingSetup(Socket, BuffStream("ping"), 10);
  ]
  If PickValid(ValueType(Socket != 8{stream}), 0) Retry;
]
Open [
  .
  .
  .
]
```

This will open a socket stream and, once the stream is open, enable the automatic transmission of the text message "ping" after every 10 seconds of no other transmission being sent.

SocketServerEnd

Description: Ends a TCP/IP socket server.

Returns: Nothing

Usage: Script Only.

Function Groups: Stream and Socket

Related to: ClientSocket | CloseStream | ServerSocket | SocketAttribs | SocketServerStart | SocketWait | SRead | SWrite | TCPIPReset

Format: SocketServerEnd(Handle)

Parameters:

Handle

Required. Any numeric expression for the socket server handle as returned from a SocketServerStart.

SocketServerStart

Description: Starts a TCP/IP or UDP socket server and returns a handle to it.

Returns: Handle

Usage: Script Only.

Function Groups: Stream and Socket

Related to: ClientSocket | CloseStream | ServerSocket | SocketAttribs | SocketServerEnd | SocketWait | SRead | SWrite | TCPIPReset

Format: SocketServerStart(Family, Port, TransmitLen, ReceiveLen, NoDelay[, ProtocolFilter])

Parameters:

Family

Required. Any numeric expression for the protocol family

Family	Description
0	TCP/IP
1	UDP

Port

Required. Any numeric expression for the port number

to offer.

TransmitLen

Required. Any numeric expression for the number of bytes to buffer when transmitting. The value must be a signed long integer, where only positive values are useful.

If the application is running on a operating system of Windows 7 / Server 2008 R2, or later, and the value is set to zero, then Windows will manage the appropriate buffer size for the link speed and latency.

If you set the buffer size, the value should match or be larger than the largest message that is expected.

A high bandwidth / high latency link will require a larger size in order to achieve optimum efficiency, but the exact size can be determined only by empirical testing..

ReceiveLen

Required. Any numeric expression for the number of bytes that VTScada will buffer internally before it stops reading from WinSock. Additional buffering is handled by WinSock.

The value must be a signed long integer, where only positive values are useful.

If the application is running on a operating system of Windows 7 / Server 2008 R2, or later, and the value is set to zero, then Windows will manage the appropriate buffer size for the link speed and latency.

If you set the buffer size, the value should match or be larger than the largest message that is expected.

A high bandwidth / high latency link will require a larger size in order to achieve optimum efficiency, but the exact size can be determined only by empirical testing.

NoDelay

Required. Any logical expression. If true, anything written to a socket started by this server will be flushed immediately. If false, packets are coalesced into larger packets to reduce network loading. This parameter should normally be false.

ProtocolFilter

An optional array of 2-element text arrays. Each 2-element array specifies a protocol which may connect to the socket. Their order in the ProtocolFilter array determines priority.
The first element in each array is the name of the protocol. The second contains the initialization string for that protocol, if required. Empty string otherwise.
The available protocols are: "SSL", "VIC", "NULL" and "PROXY".

Comments: If the parameters are all valid and the socket function fails a negative error code is returned.
If the handle to the socket is orphaned, the equivalent to a SocketServerEnd is performed
RecieveLen and NoDelay apply to streams created by inbound connections (TCP) or datagrams (UDP), not to the listener.
The handle returned can be used by the SocketWait or SocketServerEnd functions.

SocketWait

Description: Wait for Socket Connect. This function returns true when a client connects to a socket offered by a socket server.

Returns: Boolean

Usage: Script or steady state.

Function Groups: Stream and Socket

Related to: ClientSocket | CloseStream | ServerSocket | SocketAttribs | SocketServerEnd | SocketServerStart |

SRead | SWrite | TCPIPReset

Format: SocketWait(Handle)

Parameters:

Handle

Required. Any numeric expression for the socket server handle as returned from a SocketServerStart.

Comments: For a TCP connection, SocketWait triggers each time a new inbound connection is made. For UDP, SocketWait triggers each time a new datagram arrives, for which there is no existing stream to store it in. A UDP datagram is stored in an existing stream if it is from the same IP and if it is received on the same port as the other datagrams in that stream. The UDP stream can be empty (having been drained) but still be regarded as 'belonging' to the IP that instantiated the stream.

Sort

Description: Allows the sorting of an array subsection according to the order of another array.

Returns: Nothing

Usage: Script Only.

Function Groups: Array

Related to: PlotXY | TextSearch | SortArray

Format: Sort(KeyArrayElem, SortArrayElem, N, Descending [, TypeText, CaseInsensitive])

Parameters:

KeyArrayElem

Required. An element of the array to be used as the reference for the sort. This array is copied into temporary memory space. The copy is then arranged in order

along with the SortArray, but the original copy of the key array is left unchanged after the sort unless it is the same array as SortArray.
If the key array contains text strings, the TypeText parameter should be set to true (non-0) to enable alphabetical ordering, otherwise each element will be converted to a number before use as a sort key.

SortArrayElem

Required. Any array element giving the starting point in the array for the reordering. The subscript for the array may be any numeric expression. If processing a multidimensional array, the usual rules apply to decide which dimension should be used.
The new order for SortArray will be according to the order of the ordered copy of KeyArray. The values in this array may be of any type.

N

Required. Any numeric expression giving the number of array elements to use in the sort. If this number is greater than either of the two array sizes, the number of elements used in the sort will be the lesser of the two array sizes. The maximum value for N is the size of the arrays.

Descending

Required. Any numeric expression that indicates the ordering sequence to use in the sort. If it is true (non-0), the sort will be in descending (decreasing) order. If it is false (0), the sort will be in ascending (increasing) order.

TypeText

Optional. A numeric expression that controls the type of sort according to the following table. Defaults to zero - numeric sorting.

TypeText	Sort Performed
0	Numeric sort
1	Alphabetic sort
2	Tag hierarchy sort (Sorts tag names based on hierarchy rather than using a pure alphabetic sort.)

CaseInsensitive

An optional parameter that accepts any logical expression. If CaseInsensitive resolves to true (1), the key array is treated as holding case-insensitive strings, thus allowing the caller to sort alphabetically instead of lexically. The default behavior is to sort case-sensitive (lexical sort). This parameter is ignored if TypeText is false (0).

Comments:

Sort allows the re-ordering of a group of related arrays according to the order of a key array. If KeyArray and SortArray are the same array, the array is arranged in order.

This statement performs a "partition sort", using an element of the array as the dividing point, such that all other elements are divided into those greater than and less than the dividing element. Each partition is then recursively sorted. As of VTS version 7.5, the first and middle elements of the array are swapped, making the middle element the partitioning element. This has increased performance gains for arrays whose sizes reach 100,000 or more elements.

This statement is useful when used with PlotXY. For example, if there are two arrays, one having X-value and the other Y-values, of 100 data values that need to be viewed plotted against each other, the Sort statement could re-order the arrays so that the X values increased

from left to right and the Y values stayed with their corresponding X values (see example).
The order of the statements is significant. If the sort of the X array were done first, the sort of the Y array would have no effect since the X array would already be in order. In general, the key array should be sorted last.

The statement uses a minimum of eight bytes of temporary memory for each numeric array element in the sort, or more if the element has been declared to be text. This means that at least 8 * N bytes of memory is required, which may be of concern if N is large and the amount of free memory is minimal.
Invalid array entries in the key array are grouped to the end of the array.

Be aware that sorting is a relatively time-consuming operation. It should not be active during time-critical control, etc.

Example:

```
If MatchKeys(2, "sort");
[
  Sort(X[0], Y[0], 100, 0) { Sort the Y values first };
  Sort(X[0], X[0], 100, 0) { Sort the X values };
]
```

The two arrays mentioned in the previous section are sorted in preparation for plotting by the script, which is executed when the user types in "sort" from the keyboard.

SortArray

Description: Sorts an array of arrays based upon the key information provided by the second parameter. The array is sorted in-place.

Returns: Nothing

Usage: Script Only.

Function Groups: Array

Related to: Sort

Format: SortArray(array, [control, start, end]);

Parameters:

Array

Required. The array to be sorted.
The array must contain values, each of which must be either an array of simple values (numbers or text), or a record in the form of an array itself.

Control

Optional. Sets the column to be sorted and the type of sort to be performed. See the Examples section of this topic.

```
\System\SortKeys Struct [
 Column;
 Type;
 Descending;
];
```

Defaults to, Column: 0, Type: Numeric, Descending: FALSE
Note that SortKeys is defined in the system library, therefore must be preceded by \System\.

Start

Optional. The array index where sorting is to begin. If not specified, array is sorted from the first element.

End

Optional. The last array element to sort. If not specified, array is sorted to the last element. No sorting will be done if End comes before Start.

Comments: This function rapidly sorts vectors of records based upon one or more keys that can be distributed across the records.
The Array parameter cannot be a multi-dimensional array. No sort will happen if you attempt to process a multi-dimensional array.

The elements of SortKeys are as follows:
{0} Column. The field index where the key is located within each record.
{1} Type:

0 => Numeric,
1 => Case insensitive text
2 => Case sensitive text or raw binary.

{2} Descending: TRUE to sort from greatest to least. FALSE to sort from least to greatest.
The value in a field is cast to the selected Type for the purposes of the sort, but is unchanged in the resulting array.
The SortKeys parameter may be an array of SortKey structures. In this case, the array is sorted by multiple keys simultaneously, with the first SortKeys element representing the highest-priority key and so on. This is shown in the third example.
Invalid Data[i][Column] values would be sorted to the end (i.e. higher index values) regardless of the value of Descending.

Examples:

Sort on column 0 of each element of the Data array in numeric ascending order.

```
SortArray(Data);
```

Sort on column 0 case-insensitive, binary comparison, in ascending order:

```
SortArray(Data, \System\SortKeys(0, 2, FALSE));
```

Sort on column 0 numerically in descending order as primary key and secondarily sort numerically on column 1 in ascending order

```
SortInfo = New(2);
SortInfo[0] = \System\SortKeys(0, 0, TRUE);
SortInfo[1] = 1;
SortArray(Data, SortInfo);
```

Sound

Description: Plays a multimedia sound file as installed in the operating system.

Returns: Nothing

Usage: Script Only.

Function Groups: Speech and Sound

Related to: Beep | ModemDev | Play

Format: Sound(File, Option [, DevID])

Parameters:

File

Required. Any text expression giving the file name to play. If the extension is omitted, the default extension ".WAV" is added. If an empty string is provided here, then any currently playing sound is stopped.

Option

Required. Any numeric expression, which indicates how to play the file. Option is found by adding together numbers from the following table.

Option	Bit No.	How to play file
1	0	Play asynchronously (don't wait)
2	1	Don't use default sound if file missing
4	2	Reserved for future use
8	3	Loop the sound until next Sound function executed
16	4	Don't stop any currently playing sound

If Option is 0, VTScada will halt all execution until the sound is finished (this isn't recommended). Add 1 to avoid this.

DevID

An optional parameter. This parameter is required when it is desired to play a sound through other than the system default audio device. The function ModemDev can return the identifier of the wave device for a voice modem. If it is required to stop a sound playing through a specific DevID, then the value returned by the Sound function when the sound was started must be given here.

Comments:

This statement may only appear in a script.
When a sound is started using the DevID parameter, only option value 8 is significant.
If a sound is started using the DevID parameter, then the return value from the function is required in order to stop that sound.

Examples:

```
Sound("TRAIN", 1);
```

This statement plays the sound file TRAIN.WAV on the system default audio device.

```
Sound("", 1);
```

This statement stops any sound currently playing on the system default audio device.

```
SoundHandle = Sound("Irritating Music", 8, ModemDev(MStream));
```

This statement plays the sound file IRRITATING MUSIC.WAV through the audio channel of the modem whose open stream is MStream. The sound will play until stopped.

```
Sound("", 0, SoundHandle);
```

This example stops the Irritating Music sound started by the previous example

Spawn

Description: Runs another Windows™ program.

Returns: Nothing

Usage: Script Only.

Function Groups: Software and Hardware

Related to: DLL

Format: Spawn(Command)

Parameters:

Command

Required. The text string that will launch a program. If the name of a file is provided, the program associated with the given file extension will be launched. Similarly, if a URL is provided, a web browser will be launched.

Command should also contain any parameters necessary for the execution of the program. These parameters must be delimited from the command and other parameters by one or more space characters. If the command contains spaces, the command must be surrounded in double quote characters. (see examples).
If you wish to include environment variables in your command, you must spawn the command processor, CMD to run that command.

Comments: Using the Spawn statement does not stop the execution of VTScada.

> Note: Within an Anywhere Client session, this function does nothing.

Example:

```
Spawn("C:\Test Code\RunMe.BAT 1 2");
```

will be interpreted as the command C:\Test with the parameters Code\RunMe.BAT, 1 and 2.

```
Spawn("""C:\Test Code\RunMe.BAT"" 1 2");
```

will be interpreted as the command "C:\Test Code\RunMe.BAT" with the parameters 1 and 2.

```
Spawn("cmd /c ""dir 2>&1 >%TEMP%\Dirlist.txt"" ");
```

will redirect the directory contents of the current folder to a file named Dirlist.txt in the Windows temp folder, with errors (2) redirected to stdout (&1).

Speak

(VoiceTalk Module)

Description: Executes on the speech thread to speak the supplied text through a specified SAPI text-to-speech stream.

Returns: Nothing

Usage: Script Only.

Function Groups: Speech and Sound

Related to: Configure | GetDevices | GetVoices | Reset | ShowLexicon | VoiceTalk

Format: VoiceTalkStream\Speak(Text [, Flags])

Parameters:

VoiceTalkStream

Required. A speech stream returned from VoiceTalk that you wish to speak the given phrase.

Text

Required. Any text expression that will be spoken on a specified stream.

Flags

An optional parameter to specify speaking flags to the stream. All text will be spoken asynchronously (i.e. the function will not wait for the speech to complete speaking). Flags can be used to specify other parsing of the text. The values for Flags can be any combination of the following:

Flags	Meaning
0	Use default settings (speak asynchronously)
1	Speak asynchronously.
2	Purge speaking queue before speaking text. This cancels all pending and current speech, and then immediately begins speaking the new text.
4	Regard the Text parameter as a filename, and speak the contents of that file.
8	Parse text for XML markup.
16	Do not parse text for XML markup.
32	Any XML state changes in the text will persist across any future VoiceTalk\Speak calls.
64	Punctuation characters should be spoken (i.e. "Hello, there." would be spoken as "Hello comma there period").

(As indicated above, an Invalid value, or a value of "0" or "1" for the Flags parameter will have the same result.)

Comments: This function returns the error code resulting from issuing

the command to the speech engine, or zero if no error was encountered.

This function will execute and immediately return, sending the text to the speech engine to be spoken asynchronously. Asynchronous speech will not block the calling thread. You can determine when a section of text has completed speaking by inserting bookmarks into the text that can then be watched for in the VoiceTalk BookmarkNum parameter.

Multiple VoiceTalk\Speak statements can be issued on SAPI text-to-speech streams without blocking. The text will be queued up and will be spoken in the order it is submitted. The text speech queue will be terminated immediately if the SAPI text-to-speech stream is closed.

If speaking simultaneously on several streams, all to the same output device, the thread that is currently speaking will continue to do so until all its queued text has been spoken, even if the other thread issues VoiceTalk\Speak calls during this time.

The text string spoken can optionally contain embedded "control tags" that affect the way that the text is spoken. These control tags are in the form of embedded XML. For example, <EMPH> and </EMPH> emphasizes the words between the tags, and <BOOKMARK MARK="32"/> sets the SAPI text-to-speech stream current bookmark number to 32. In order for embedded XML to be parsed, either the text stream must be enabled to process XML (by passing a value for Flags that includes the value 8), or the text string itself must begin with an angle bracket (<).

For a complete list of embedded control tags, please refer to the Microsoft SAPI 5.1 speech documentation.

Example:

```
sHandle = \VoiceTalk();
If Valid(sHandle) && ZButton(10, 40, 110, 10, "Talk", 1);
[
```

```
  sHandle\Speak("<P>Mary had a <EMPH>little</EMPH> lamb</P>");
]
```

This will speak the supplied text with emphasis on the word "little".

```
sHandle = \VoiceTalk();
If Valid(sHandle) && Value < 0 Warning;
[
  sHandle\Speak("The value has fallen below zero", 1+2 {synchronous &
purge});
]
```

This will immediately stop any current speech, begin speaking the warning to the user, and continue on right away to the state Warning.

SpeakToFile

(VoiceTalk Module)

Description: Executes on the speech thread to speak the supplied text to a .wav format audio file.

Returns: Nothing

Usage: Script Only.

Function Groups: Speech and Sound, File I/O

Related to: Configure | GetDevices | GetVoices | Reset | ShowLexicon | Speak | VoiceTalk

Format: VoiceTalkStream\SpeakToFile(Phrase, Flags, Filename, Quality[, Result])

Parameters:

VoiceTalkStream

Required. A speech stream returned from VoiceTalk that you wish to speak the given phrase.

Phrase

Required. Any text expression that will be spoken on a specified stream.

Flags

Required. A parameter to specify speaking flags to the stream. All text will be spoken asynchronously (i.e. the function will not wait for the speech to complete speaking). Flags can be used to specify other parsing of the text. The values for Flags can be any combination of the following

Flags	Meaning
0	Use default settings (speak asynchronously)
1	Speak asynchronously.
2	Purge speaking queue before speaking text. This cancels all pending and current speech, and then immediately begins speaking the new text.
4	Regard the Text parameter as a filename, and speak the contents of that file.
8	Parse text for XML markup.
16	Do not parse text for XML markup.
32	Any XML state changes in the text will persist across any future VoiceTalk\Speak calls.
64	Punctuation characters should be spoken (i.e. "Hello, there." would be spoken as "Hello comma there period").

As indicated above, an Invalid value, or a value of "0" or "1" for the Flags parameter will have the same result.

Filename

Required. The name of the .wav file to speak into (e.g.

"c:\folder\file.wav")

Quality

Quality	Format	Quality	Format
-1	SAFTdefault		
4	SAFT8kHz8BitMono	5	SAFT8kHz8BitStereo
6	SAFT8kHz16BitMono	7	SAFT8kHz16BitStereo
8	SAFT11kHz8BitMono	9	SAFT11kHz8BitStereo
10	SAFT11kHz16BitMono	11	SAFT11kHz16BitStereo
12	SAFT12kHz8BitMono	13	SAFT12kHz8BitStereo
14	SAFT12kHz16BitMono	15	SAFT12kHz16BitStereo
16	SAFT16kHz8BitMono	17	SAFT16kHz8BitStereo
18	SAFT16kHz16BitMono	19	SAFT16kHz16BitStereo
20	SAFT22kHz8BitMono	21	SAFT22kHz8BitStereo
22	SAFT22kHz16BitMono	23	SAFT22kHz16BitStereo
24	SAFT24kHz8BitMono	25	SAFT24kHz8BitStereo
26	SAFT24kHz16BitMono	27	SAFT24kHz16BitStereo
28	SAFT32kHz8BitMono	29	SAFT32kHz8BitStereo
30	SAFT32kHz16BitMono	31	SAFT32kHz16BitStereo

Result

Required. A pointer to a flag to set when done. This should be 0 for OK or 1 for an error.

Comments: The text may include embedded XML tags as described in the documentation for the SAPI speech engine being used. Multiple calls to this function will be queued.
Calls to this module wil l fail if the VoiceTalk instance was not configured for VTSFileOutput.

Spinbox

(System Library)

Description: Draws a spinbox with optional label.

Returns: Nothing

Usage: Steady State only.

Function Groups: Graphics

Related to: Bevel | Boolean | CheckBox | CheckFileExist | CheckPathExist | ColorSelect | CopyDir | Debugger | DialogInitPos | Droplist | Edit | Folder | GridList | HScrollbar | Listbox | RadioButtons | ReadINI | ReadSectINI | SplitList | ToolBar | VScrollbar | WriteINI | WriteSectINI |

Format: \System\Spinbox(X1, Y1, X2, Y2, Variable, Label, BoxOnLeft, Alignment, NumChars, LowLimit, HighLimit [, CanEdit, FocusID, TextOption, TextValue, Trigger, BGCo-lor, FGColor, WidthOut])

Parameters:

X1

Required. Any numeric expression giving the X coordinate on the screen of one side of the spinbox and its label.

Y1

Required. Any numeric expression giving the Y coordinate on the screen of either the top or bottom of the spinbox.

X2

Required. Any numeric expression giving the X coordinate on the screen of the side of the spinbox and its label opposite to X1.

Y2

Required. Any numeric expression giving the Y coordinate on the screen of the top or bottom of the spinbox, whichever is the opposite to Y1.

Variable

Required. The variable whose value is set by the spin-box.

Label

Required. Any text expression to be used as a label with the spinbox.

BoxOnLeft

Required. Any logical expression. If true (non-0) the spinbox will appear to the left of the label, if false (0) it will be to the right. If this value is invalid, a default value of true will be used.

Alignment

Required. Any numeric expression that sets the alignment of the spinbox and its label according to one of the following options: The default value is 0.

Alignment	Horizontal	Vertical
0	Left	Top
1	Right	Top
2	Full	Top
3	Left	Centered
4	Right	Centered
5	Full	Centered
6	Left	Bottom
7	Right	Bottom
8	Full	Bottom

NumChars

Required. Any numeric expression that gives the number of digits wide to make the spinbox. A value of 0 or invalid results in the spinbox being automatically sized to fit the widest number (or text string if TextOption and TextValue are set).

LowLimit

Required. Any numeric expression giving the lowest permissible value. If the spinbox is editable and a value less than LowLimit is entered, it will revert to the LowLimit value.

HighLimit

Required. Any numeric expression giving the highest permissible value. If the spinbox is editable and a value greater than HighLimit is entered, it will revert to the HighLimit value.

CanEdit

An optional parameter that is any logical expression. If true (non-0), the number in the field may be edited directly, if false (0), it may not. The default value is false.Note that the value of this parameter directly affects the TextOption and TextValue parameters' effectiveness. If CanEdit is true, both are ignored.

FocusID

An optional parameter that is any numeric expression for the focus number of this graphic. If this value is 0, the spinbox will display its current setting, but the value will not be able to be set and the spinbox field will appear grayed out.If this parameter is omitted, keyboard input (such as the arrow keys) will be ignored. The default value is 1.

TextOption

An optional parameter that is any text expression used to replace a certain value (expressed by TextValue) in the spinbox field. This parameter will be ignored if CanEdit is true.

TextValue

An optional parameter that is any numeric expression for the value in the spinbox that is to be replaced by the text string in TextOption. This parameter will be ignored if CanEdit is true.

Trigger

An optional parameter that is a numeric expression. The value in Trigger will become 0 if the user changes the internal buffer (i.e. when the value of the WinEditCtrl as logged in the variable Change transits from invalid to zero).
If the user presses any of Enter, the spin box arrows or the arrow buttons on the keyboard, Trigger becomes 1. If the spinbox loses focus, the value of Trigger becomes 2.

BGColor

Optional. Any numeric expression for the background color of the control. No default value.

FGColor

Optional. Any numeric expression for the foreground color of the control. No default value.

WidthOut

Output. The overall width required for the control. Includes space required for the displayed characters, borders and spin button.

Comments:

This module is a member of the System Library, and must therefore be prefaced by \System\, as shown in the "Format" section. If you are developing a script application, use "System\..." rather than "\System\..." in the function call.

The size of the spinbox is constant, with X1, Y1and X2, Y2 defining the position of the check box and its label.

For any optional parameter that is to be set, all optional parameters preceding the desired one must be present, although they may be invalid.

The trigger parameter provides an indication that something has changed and therefore the user should be prompted to save changes before exiting.

Example:

```
System\Spinbox(200, 70, 360, 30 { Location of spinbox },
               Retries { Variable to change },
               "Number of retries"{ Label },
               Invalid { Defaults to box on left },
               6 { Left, bottom align },
               4 { Field width in chars },
               0, 100 { Low, high limits },
               0 { Not editable },
               3 { Focus ID },
               "None" { Text replacement },
               0 { Use text in index 0 });
```

SplitList

(System Library)

Description: Draws a split list (listbox with two columns) with a scrollbar if required and indicates the selected item.

Returns: Nothing

Usage: Steady State only.

Function Groups: Graphics

Related to: Bevel | CheckBox | Droplist | GridList | HScrollbar | Listbox | RadioButtons | Spinbox | ToolBar | VScrollbar

Format: \System\SplitList(X1, Y1, X2, Y2, Title1, Title2, Data1, Data2, Index, Picked[, Flat, DoubleClick, MaxLen, Offset, FocusID, AlignTitle, Multi, PickList, SplitPos, ShrinkData1CBContext, ShrinkData1Callback, BGColor, FGColor])

Parameters:

X1

Required. Any numeric expression giving the X coordinate on the screen of one side of the listbox.

Y1

Required. Any numeric expression giving the Y coordinate on the screen of either the top or bottom of the listbox.

X2

Required. Any numeric expression giving the X coordinate on the screen of the side of the listbox opposite to X1.

Y2

Required. Any numeric expression giving the Y coordinate on the screen of the top or bottom of the listbox, whichever is the opposite to Y1.

Title1

Required. Any text expression giving the title to be displayed above the first column of the split list.

Title2

Required. Any text expression giving the title to be displayed above the second column of the split list.

Data1

Required. An array of data to be displayed in the first column of the split list. This array must be the same size as Data2.

Data2

Required. An array of data to be displayed in the second column of the split list. This array must be the same size as Data1.

Index

Required. Any variable whose value will be set to the index of the highlight.

Picked

Required. A variable whose value will be set true (1) when an item is chosen in the split list. The setting of Index by an external source will not trigger Picked.

Flat

Any logical expression. If true (non-0) the border of the split list will be a single black line, if false (0) it have the look of two indented windows. Defaults to 0.

DoubleClick

An optional parameter that is a variable whose value will be set to true (1) when an item has been double-clicked upon.

MaxLen

An optional parameter that is any numeric expression giving the maximum length of the data lists. If omit-

ted, the maximum list length is given by the size of the arrays Data1 and Data2.

Offset

An optional parameter that is any numeric expression giving the starting offset in the list. The element indicated by this index will be the one initially shown at the top of the list (unless too few elements follow this one to fill the display area).

FocusID

An optional parameter that is any numeric expression for the focus number of this graphic. If this value is 0, the split list will not accept keyboard input, although mouse input will still be recognized. The default value is 1.

AlignTitle

An optional parameter that is any logical expression. If true (non-0) the title is drawn within the split list's boundaries, if false(0) the list fills its bounding area and the title is added at the top (i.e. it extends past the top boundary). The default is true.

Multi

An optional parameter that is any logical expression. If true (non-0), multiple items may be selected in the list. The default is false.

PickList

An optional parameter that is a variable whose value is set to the list of items selected if Multi is true (1). If invalid, no items are selected. This variable may initially be set to a dynamically allocated array (one created with the New function) containing items to be highlighted/selected upon the startup of the split list.

SplitPos

An optional parameter that is a variable that will con-

tain the pixel position of the split bar. If initially valid, the split bar will be in the middle of the SplitList. The default will center the split bar.

ShrinkData1CBContext

Context for callback to shorten Data1 text

ShrinkData1Callback

Callback(Text, AvailWidth, Font)

BGColor

Optional. Any numeric expression for the background color of the control. No default value.

FGColor

Optional. Any numeric expression for the foreground color of the control. No default value.

Comments:

This module is a member of the System Library, and must therefore be prefaced by \System\, as shown in the "Format" section. If you are developing a script application, use "System\..." rather than "\System\..." in the function call.

If Multi is true, multiple items in the list may be selected by using the <Shift> or <Ctrl> keys along with mouse input. If <Ctrl> is held while an item is clicked on by the mouse, it will become selected (or de-selected if it is already selected) and will be added to the list of chosen items. If <Shift> is held while an item is clicked on by the mouse, all items from the last selected item to the currently selected item will be selected. All other items outside of this list will be de-selected. If both <Ctrl> and <Shift> are held while an item is clicked on by the mouse, all items from the last selected item to the currently selected item will be set to the state of the last selected item.

For any optional parameter that is to be set, all optional parameters preceding the desired one must be present, although they may be invalid.

Examples:

```
System\SplitList(20, 325, 340, 50 { Outline of splitlist },
                 NameLabel { Column 1 title },
                 TypeLabel { Column 2 title },
                 NameTable { Column 1 data },
                 TypeTable { Column 2 data },
                 Index { Highlighted element },
                 0 { Picked not required },
                 0 { 3D look },
                 DClick { Set for double click },
                 Invalid { Use default max length },
                 0 { Start at top of list },
                 3 { Focus ID },
                 0 { Title extends past top });
```

SplitListSelector

(System Library)

Description: Draws a split view comprising two GridLists separated by control buttons. Items listed on the left may be selected to be transferred to the right. Items on the right may be returned to the left so long as they were originally listed on the left. Scrollbars will be drawn if required.

Returns: Nothing

Usage: Steady State only.

Function Groups: Graphics

Related to: SplitTagSelector | GridList

Format: \System\SplitListSelector(InitKeyArray, DestKeyArray[, LeftLabel, RightLabel, NumCols, Titles, ColWidthPercents, DestMax, EnableParm])

Parameters:

InitKeyArray

Required. An array of items passed in to form the initial list on the left.

DestKeyArray

Required. An array of destination items. Any items initially in the array will be displayed in the list on the

right.

LeftLabel

Optional label for the left list.

RightLabel

Optional label for the right list.

NumCols

Number of columns for each list. Defaults to 1 if not specified.

Titles

Optional array of titles for both lists. The number of titles must match the number of columns. If this parameter is not specified, then no titles will be displayed for the columns.

ColWidthPercents

Optional array of column width percentages for both lists. Defaults to equal widths if not specified. The size of the array must match the number of columns. The array elements will hold normalized values. For example:

```
ColWidths    = New(3)
ColWidths[0] = .3 {30%}
ColWidths[1] = .2 {20%}
ColWidths[2] = .5 {50%}
```

DestMax

Maximum number of destination items. The default is invalid which is taken to mean unlimited.

EnableParm

Flag to enable controls. Defaults to 1 if not specified

Comments:

A tool intended to provide a user interface for selecting items from a list on the left and transferring them to the right. The InitKeyArray may be an array of arrays. i.e. each

element of the main array may be a sub-array which will make up the columns. If an array of arrays is specified, the DestKeyArray must also be of the same format.
The first column (the first element of every sub-array making up a row) will be regarded as the unique key for comparison purposes. The items on the left will be updated if the InitKeyArray changes in size.
When items are moved from the right to the left, they are inserted into the left only if they exist in the InitKeyArray. If not, they are simply removed from the right list.
The SplitListSelector should be called from within a GUITransform as shown in the following example.

Examples:

```
GUITransform(30, 270, 470, 90,
             1, 1, 1, 1, 1 { Scaling              },
             0, 0          { Movement             },
             1, 0          { Visibility, scaling  },
             0, 0, 0       { Selectability        },
             \System\SplitListSelector(InitResourceList,
                                SelectedResources,
                                  \AvailableResourcesLabel,
                                  \SelectedResourcesLabel));
```

SplitPath

Description: Breaks up a file path name into its components.

Returns: Nothing (path components are returned in the parameters)

Usage: Script or steady state.

Function Groups: File I/O

Related to: Dir | FileFind

Format: SplitPath(FullName, Drive, Path, File, Extension)

Parameters:

FullName

Required. Any text expression giving the full path name.

Drive

Required. Any variable, which is set to the drive letter in FullName, plus a colon. If FullName doesn't contain a drive specification, Drive is a null string. If this information is not required, a constant may be used for this parameter.

Path

Required. Any variable, which is set to the directory path in FullName, ending in a backslash. If FullName doesn't contain a path specification, Path is a null string. If this information is not required, a constant may be used for this parameter.

File

Required. Any variable, which is set to the file name in FullName. If FullName doesn't specify a file name, File is a null string. If this information is not required, a constant may be used for this parameter.

Extension

Required. Any variable, which is set to the file extension in FullName, including a period). If FullName doesn't specify a file extension, Extension is a null string. If this information is not required, a constant may be used for this parameter.

Examples:

```
SplitPath("C:\MyApps\App1\Info.TXT", drv, pth, fName, ext);
```

The values of drv, pth, fName and ext will be "C:", "\MyApps\App1\", "Info" and ".TXT" respectively.

```
SplitPath("D:\Apps\Profiler\GDI.WIF", drv, pth, 0, 0);
appPath = Concat(drv, pth);
```

Note the use of "0" for parameters that are not required.

SplitTagSelector

Description: Draws a split view comprising two GridLists separated by control buttons. Tag names listed on the left may be selected to be transferred to the right. Each GridList will have two columns: the name and the description of each tag in the list. Scrollbars will be drawn if required.

Returns: Nothing

Usage: Steady State only.

Function Groups: Graphics

Related to: SplitListSelector | GridList

Format: \SplitTagSelector(InitTagArray, DestTagArray[, LeftLabel, RightLabel, DestMax, EnableParm])

Parameters:

InitTagArray

Required. An array of tag names passed in to form the initial list on the left.

DestTagArray

Required. Array of destination tag names. Can be an array of tags passed in to form the initial list on the right.

LeftLabel

Optional label for the left list.

RightLabel

Optional label for the right list.

DestMax

Maximum number of destination tag names. The default is invalid which is taken to mean unlimited.

EnableParm

Flag to enable the controls.

Comments: A tool intended to provide a user interface for selecting tags from a list on the left and transferring them to the right.
Both GridLists have two columns, the name and the description of the tags. The InitTagArray is an array of Tag names.
The DestTagArray is an array of selected Tag names. The Tag names are used as keys for comparison purposes. The tags on the left will be updated if the InitTagArray changes. When tags are moved from the right to the left, they are inserted in the left only if they exist in the InitTagArray. If not, they are simply removed from the DestTagArray.
The SplitTagSelector should be called from within a GUITransform as shown in the following example.

Examples:

```
GUITransform(30, 270, 470, 90,
             1, 1, 1, 1, 1 { Scaling              },
             0, 0          { Movement             },
             1, 0          { Visibility, scaling  },
             0, 0, 0       { Selectability        },
             \SplitTagSelector(InitResourceList,
                               SelectedResources,
                                 \AvailableResourcesLabel,
                                 \SelectedResourcesLabel));
```

SQLQuery

(VTSSQLInterface library)

Description: A launched module that executes an SQL query on data in a VTS application.

Returns: Nothing
(The parameters will be populated with pointers to the returned information)

Usage: Script Only.

Function Groups: Database and Data Source, ODBC

Related to: RegisterCustomTable

Format: \VTSSQLInterface\SQLQuery(QueryString, Results, FieldNames, FieldTypes, ReturnCode, ErrorMsg)

Parameters:

QueryString

Required. Any text expression containing a valid SQL select statement. Not all selection clauses are supported.

Results

Required. A pointer to a variable which will hold the results array.

FieldNames

Required. A pointer to a variable that will hold the field names array.

FieldTypes

Required. A pointer to a variable that will hold the field types. The types will be returned as SQL data types according to the following table:

Field Type	SQL Data Type
1	SQL_CHAR
4	SQL_INTEGER
5	SQL_SMALLINT
8	SQL_DOUBLE
9	SQL_DATETIME
12	SQL_VARCHAR

ReturnCode

Required. A pointer to a variable that will hold the return code. The code will be one of the following:

Return Code	Meaning

0	#SUCCESS
1	#SYNTAX_ERROR
2	#TABLE_NOT_FOUND
3	#COLUMN_NOT_FOUND
4	#ILLEGAL_JOIN

ErrorMsg

Required. A pointer to a variable that will hold a textual error message. Valid only if ReturnCode is not zero.

Comments:

Only selection queries are supported. SQL statements for data manipulation will do nothing. If tag filtering or realm-area filter is in effect, this function will retrieve data only from tags the currently logged-on user is permitted to access.

When retrieving historical data, SQLQuery is essentially a wrapper for GetTagHistory. It takes an incoming SQL query, and makes one or more calls to GetTagHistory to retrieve the results. SQLQuery may also be used to retrieve current tag values and other custom tables (such as alarm data).

Legacy tables made it look like tag values were stored in separate tables and used the time stamp of the current server. These legacy tables still exist but are hidden by default, using the property: SQLQueryHideLegacyTables. Newer code should query all tag values from the table, "History", or from a "_TPP"derivative such as "History_1d". See the links for further information.

Supported SQL syntax is as follows:

SELECT [DISTINCT | ALL] columnspecifier-1, columnspecifier-2, ...

FROM tablename-1, tablename-2, ...

[WHERE where-expression]

[ORDER BY columnspecifier-1 [ASC | DESC], columnspecifier-2, ...]
[LIMIT [offset,] row_count]

- Columnspecifier is either [table-name.]'*' to indicate all columns in a table or a specific column-name in the form: [table-name.]column-name
- Tablename is either the table name or 'table-name [[AS] alias-name']
- Quotes may be used around table or column names, and must be used if the names contain special characters. When an alias-name is specified it may be used in place of the table-name in the column specifier.
- When more than one table is specified, the tables are automatically joined based on their Timestamp columns and MUST have matching "TPP"s. A join expression may be used in place of a table list, but it is only parsed to extract the tables specified; the actual join expression is ignored.
- Where-expression is an expression to be used to filter the result data.
- It may contain references to columns, use comparison operators, use functions ABS, LENGTH, UPPER, LOWER, CONCAT, CASE, SQRT, INTEGER, and use keywords AND, NOT, and OR.
- Use the LIMIT clause to specify a limit to the number of rows that will be returned, and optionally an offset (0-based) which allows pagination. LIMIT is applied after WHERE filtering and after sorting.
- If there is no explicit LIMIT clause in the query there is an implicit "LIMIT SQLQueryMaxResultRows" (Settings.Dynamic setting) added.

Each logged tag corresponds to one table in the VTS database schema.

Every logged variable within the tag is one column. In addition, a "TPP specifier" may be appended to a tag name to utilize GetTagHistory's ability to retrieve data over time periods. The TPP specifier is an underscore followed by a number and an abbreviation for various time periods. The recognized time period abbreviations are (case-insensitive):

- MS – milliseconds
- S – seconds (this is the default; the S may be left out)
- M – minutes
- H – hours
- D – days
- W – weeks
- Y – years

For example, the table 'ai1_2D' can be used to retrieve data from tag ai1 with a TPP of 2 days.
A TPP may only be specified for a tag that has at least one numeric logged variable. When a TPP is specified, the available columns are:

- Timestamp
- VarName:Average
- VarName:Minimum
- Varname:Maximum
- VarName:Delta
- VarName:ValueAtStart
- VarName:TimeOfMin
- VarName:TimeOfMax
- VarName:ZToNZCount
- VarName:NonZeroTime
- VarName:Total
- VarName:Interpolate

... where VarName may be replaced by any numeric

logged variable from the tag.
The current values of a tag may be retrieved using a table name with the format "TagName_Current". This table will have one row whose columns are Timestamp (the current time) and the current values of each logged value in the tag.

Related Information:

... See: "VTScada SQLInterface Module" in the VTScada Programmer's Guide
The following can be found in the VTScada Developer's Guide:

...Data Available to the ODBC Interface

...SQL Queries: Reference & Examples

Sqrt

Description: Returns the square root of a number.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Generic Math

Related to: Exp | Ln | Log | Pow

Format: Sqrt(X)

Parameters:

X

Required. Any numeric expression giving the number to take the square root of.

Comments: If X is less than zero, the result is invalid.

Example:

```
rootNum = Sqrt(25);
```

The variable rootNum will be set to 5.

SRead

Description: Reads values from a formatted stream and returns the number of values not read.

Returns: Numeric

Usage: Script Only.

Function Groups: Stream and Socket

Related to: BuffRead | CloseStream | FileSize | FileStream | FRead | GetStreamLength | StreamEnd | SWrite

Format: SRead(Stream, Format, V1, V2, V3, ...)

Parameters:

Stream

Required. Any expression that returns the stream to read.

Format

Required. Any text expression giving the format of how the values (Vn parameters) are to be read.

This format is similar, but not identical, to the C language format string for the scanf function, whereby each of the % format specifications assigns a value to one of the Vn parameters in the statement in the order in which each appears in the list.

Note that like a standard text string, these format specifiers must also be enclosed by double quotes. If a format specification appears for which there are no remaining V parameters, the format specification value is read and discarded.

For the % format specifications, the following form applies (where the [ ] indicates optional elements):
%[*][width]type

Where...
% is mandatory;
The optional asterisk ***** causes the read to occur as per the format specification, but suppresses any assignment to the Vn parameters; and
width is mandatory, specifying the maximum number of characters to read.
The specifications for **type** are listed in the following table:

Note: Note: Format strings are case insensitive. Additionally, specifying a character for a type that is not in this list results in all the characters following the % up to that point to be read exactly as they appear in the Format string and discarded.

Type	Meaning
Nb	Binary format, where n is a number indicating the type of value (see below)
c	Single ASCII character (byte)
d	Signed decimal integer
e	Signed exponential
f	Signed floating point
g	e or f formats
i	Signed decimal integer

l	Line of characters terminated by a carriage return, line feed, or both
n	Present offset in the buffer
o	Unsigned octal
s	Text string
u	Unsigned decimal integer
x	Unsigned hex integer using "abcdef"
znnn	Escape character where nnn is the 3-digit ASCII code

nb, Binary type For the format specification of %nb, where n specifies the type of number, n must be a single digit from one of the following choices. All are low-byte-first.

n value	Type
0	Byte
1	Short integer (2 bytes, low byte first)
2	Long integer (4 bytes, low bytes first)
3	IEEE single precision float (4 bytes)
4	<obsolete>
5	IEEE double precision float (8 bytes)
6	<obsolete>
7	Binary unsigned short (2 bytes, low byte first)
8	Unsigned 32-bit integer

c, ASCII character type: Unlike BuffWrite this type deals with characters in a string; each char-

acter being equal to one byte. Unlike the **%s** option, which reads only up to the first white-space character, the **%c** option reads the number of characters/bytes specified by its width and is not terminated by any particular character. If no width is specified, a single character is read.

d, Signed decimal integer

e, Signed exponential

f, Signed floating point

g, e or f formats

i, Signed decimal integer type: This option normally reads a decimal integer; however, if a leading "0b" is encountered, the number will be interpreted as binary. If a leading "0" (zero only) is encountered, the number will be interpreted as octal. If a leading "0x" is encountered, the number will be interpreted as hexadecimal.

l, Line of characters: This option reads a line of characters terminated by a carriage return, a line feed, or both (in either order). The carriage return and line feed will be discarded, and the next character read will be the first character on the next line. The maximum number of characters read is 4096 (or less if the width option is used).

n, Buffer offset: This option does not read a

value, but returns the present offset in Buffer and can be useful in subsequent reads.

o, Unsigned Octal

s, Text string type: Text in the string is read up until a white-space character is encountered, or the specified width has been read, whichever is smaller. Square brackets enclosing a character, group of characters, or a caret and a group of characters used in the format string reads strings not delimited by spaces. This is a substitute for the %s format specification. The input is read up to the first character that does not appear inside the square brackets (note that this is case-sensitive). A dash may be used to specify a range of characters. For example, the following format specifier:
% [A-Fa-f]
will read a string up to the first which is not an A, B, C, D, E, or F both upper and lower case.

The caret symbol ^. If the first character inside the square brackets is a caret (^), the read progresses up to, but not including, the first character that appears inside the square brackets:
%[^X-Z]
This would read a string up to, but not including, the first X, Y or Z (upper-case only); if the string were terminated by an X, the next character read would be that X. Inside the square brackets, the backslash is used as an escape character - any character following a backslash

(such as a caret, dash, or backslash) is taken as that character without special meaning. For example:
%[^X-Z\^]
would behave as described previously, except that the string would now be read up to but not including the first X, Y, Z, or ^.
Since format specifications for the Vn parameters are indicated by a percentage sign, to read (and discard) an actual percentage sign as part of the text string, precede it with a backslash character (i.e. \%). Also, since the backslash character is used in this manner, as well as with special control characters such as line feed, carriage return and form feed, to read and discard a backslash, use two backslash characters (i.e. \\).

x, Hexadecimal characters: the %x option reads the number of characters/bytes specified by its width and is not terminated by any particular character. If no width is specified, it will continue reading all bytes that can be recognized as hexadecimal characters. For example, given the string "...= 3D", %[^=]=%2x would read the hedadecimal value, 3D (decimal value, 61).

znnn, Escape characters: This specifies an escape character that will be thrown away when read, where nnn is a 3-digit number giving the ASCII character code of the escape character. This option is generally used as the sole format specifier that reads an entire string, spaces

included, discarding every single occurrence of an escape character, or the first occurrence of every pair of escape characters. For example, if the string to be read looked like:

abXc dXXfghiXXXjXXXXkl mX Xn o

and the format specifier indicated that the ASCII code for 'X' (88) was to be the escape code:

%25z088

then the variable that this was read into would contain:

abc dXfghiXjXXkl m n o

Notice that for each occurrence of X, the character immediately following it is saved, even if it is itself an escape character. Then the next occurrence of the escape character is discarded, with the character following it being saved, regardless of what it is, and so on. The width field specifies the maximum number of bytes to place in the output string; if this number is smaller than the input string (less the offending escape characters), the string will be truncated. If no width is specified, a single character will be read.

Control characters: In order to encode certain control characters as part of the Format parameter, one of two methods may be used. The

first is to use a backslash character followed by one of the single character codes listed below to produce the desired result. Please note that the letters must be lower case.

Code	Meaning
\b	Backspace
\f	Form Feed
\n	Line Feed
\r	Carriage Return
\t	Horizontal Tab
\v	Vertical Tab

In addition to the predefined codes, an alternate form may be used:

\nnn: where nnn is a three digit integer in the range of 0 to 255 specifying a certain ASCII character. If the number contains less than three digits, the leading spaces must be padded with zeroes; this is not the case with the previously listed single character control characters. For example, to include the one byte ASCII character G in the output, you could place its decimal equivalent of 71 in the Format string as \071.

V1, V2, V3

Optional. Parameters specifying the variables to be read in the form described by the Format parameter.

Expressions are not allowed.

Each of the Vn parameters is read in the order in which each appears in the parameter list. V1 has

the format given by the first % sequence in the Format parameter, V2 has the second, and so forth.

V1, V2, V3, ...

These parameters are the variables to be read in the form described by the Format parameter. Expressions are not allowed. Each of the Vn parameters is read in the order in which each appears in the parameter list. V1 has the format given by the first % sequence in the Format parameter, V2 has the second, and so on..

Comments: This function is useful for reading formatted streams. Data exchange between many formats is possible if the formats are known. The return value is optional and is the number of Vn parameters NOT read. This can be used as an error flag. This function will only read values up to a length of 4096 bytes long at which point it will truncate the value.

Examples:

```
If ! Valid(err);
[
  err = SRead(recipeStream, "%f%f%f",compoundA, compoundB,
  resin);
]
```

This reads three ASCII format floating-point numbers, from the current position in the stream in the variable recipeStream. The three numbers are placed in compoundA, compoundB and resin, respectively. If the read were successful, err is 0. Otherwise err is the number of items which were not read.

```
streamData = "Hello!World!How!Are!You!";
If ! Valid(notRead);
[
  notRead = SRead(streamData, "%[^!]!%[^!]!%[^!]!%[^!]!%[^!]!",
  word1, word2, word3, word4, word5);
]
```

This is an example of multiple fields delimited by the same character, in this case an exclamation point, that can be read by using the %[^

aCharacter] format. Notice the exclamation point following each format character – this causes the string read into each of the variables to be without its trailing exclamation point.

Start

Description:	Starts an application
Returns:	Nothing
Usage:	Steady State only.
Function Groups:	Configuration Management
Related to:	GetAppInstance | AppIsRunning | AppIsStarted | AppIsStarting |
Format:	Layer\Start()
Parameters:	none.
Comments:	To use, gather the "Layer Root" of an object from the \System\GetAppInstance function. Has no effect if the application is already running.

Examples:

The following snippet from a script application will start the Completed Tutorial application (The GUID of the Completed Tutorial may vary).

```
[
  WaitObj;
  CompLayer;
  TutGUID = "db53f244-90ef-4628-bdf6-2d53794a2079";
]
Main [
  If !Valid(WaitObj) && !Valid(CompLayer) WaitLayerLoad;
  [
    { GetAppInstance doesn't return the Layer until it has loaded. }
    WaitObj = Layer\GetAppInstance(TutGUID, &CompLayer);
  ]
]
WaitLayerLoad [
  CompLayer\Start();
]
```

StartTag

Note: Deprecated. Do not use in new code.

Use ModifyTags for all new code.

Description: Used to create tags by starting new instances of the tag type specified in the parameter list. When creating an application that requires child tags, it is recommended that this function be used in place of the older ChildLaunch function.
StartTag can also be used to stop or to modify an existing tag.

Returns: Numeric (0)

Usage: Script Only.

Function Groups: Advanced Module

Related to: ModifyTags | OpChange | SimpleOpChange

Format: StartTag(Parent, Flags, TagType, ParameterList)

Parameters:

Parent

An object that gives the parent for this new child tag. Defaults to VTSDB if invalid. See note in Comments section.

Flags

Bitwise expression indicating operational options.

Bit #	Operation
0	Add or change if TRUE. Delete if FALSE.
1	Persist change if TRUE. The tag will be output to the tag file.

If Bit 0 is FALSE, it is not strictly relevant whether Bit 1 is TRUE or FALSE. The tag will be

deleted both in memory and on disk. Regardless of this, it is good form to set bit 1 to TRUE when deleting a tag.

When using StartTag to generate child tags, do not set bit 1 to TRUE.

Flags should be set such that when the tag is stopped, StartTag is called with the a value of zero in the second parameter. This can be done using the expression, Valid(Root\Name). See: Parent Tags for examples.

TagType

Required. The name of the tag type to be used for the tag. If modifying or deleting an existing child tag, you may set this parameter to invalid.

ParameterList

Required. All the parameters required by the child tag should be provided here. The child tag's parameter names and matching values may be supplied in any one of the follow forms:

- As a comma-separated list of up to 256 name-value pairs.
- As a two dimensional array of name-value pairs
- As two arrays, the first containing the parameter names and the second parallel array, containing the matching values.
- As a dictionary.
- As a one-dimensional array of values. The order must match the order of parameters in the tag type, with no gaps or spaces. Note that there is no guarantee that the parameter order will remain the same in future versions - use of a dictionary is preferred.

While there is no requirement that the child parameters must be given in the order that they are defined in the child tag, better performance will be obtained by matching that order. It is necessary to know the names of the parameters.

Comments:

When used to create a child tag, the name of that new child tag will always be unique since it will be a combination of the parent tag's name and the new child name in the format "ParentName\ChildName".

If the first parameter (Parent) is not VTSDB, then it should be a tag object (the parent tag), and the StartTag call should only be made from that parent tag's Refresh module.

When this function is used to create an ordinary tag, as opposed to a child tag, it is the responsibility of the programmer to ensure that a unique name is used for each new tag.

- If the tag does not exist and bit 0 of Flags is TRUE, then the tag is launched.
- If the tag exists and bit 0 of Flags is TRUE, then the tag's parameters are changed if necessary. Note that, if a new TagType is provided, then the existing tag will be deleted and a tag of the new type created.
- If the tag exists and bit 0 of Flags is FALSE or the tag's parent's Name is Invalid indicating that the tag is being stopped, then the tag is removed.

Because of configuration management and the version control system, it is possible for a child tag have both temporary parameters (those created by code in the parent and existing only in memory) and a sub-set of permanent parameters in the tag files. These latter values are used to record user-overrides of the parameter values. If the application

restarts, the child tag will be re-loaded from code and the overrides made in an earlier session will be read from the tag files. Permanent parameter values will always take precedence over temporary values. There are several ways that the tag's parameters can be passed to this module:

- Only one additional parameter which is an array of parameter values, a tag, or a tag mirror structure(*). In the case of a parameter value array the values must match to tags parameters in order, but may be fewer in number, allowing just the name to be passed, for example.
- Only one additional parameter which is a dictionary of parameter values keyed by parameter name.
- A pair of arrays is passed, with the first array being the array of parameter names and the second, parallel array, being the values for those named parameters. The "Name" parameter must be present, and the order of the pairs is not significant, other than matching the formal parameter ordering gives some efficiency gains.
- A list of parameter name and value pairs of parameters. This is typically used by ChildLaunch calls. Again "name" must be present and the ordering is only a small consideration for efficiency.

(*) A tag mirror is a structure that has one element for each of a tag's parameters, accessible by that parameter name (e.g. MyTag\IODevice). These should only be used by advanced VTScada programmers.

In all examples, ParentRoot is assumed to be a pre-existing tag object with name "Something".

Examples:

Example 1 – Creating a child tag by supplying a series of name, value pairs.

```
Code\StartTag(ParentRoot, 0b11, "AnalogStatus",
       "Name",         "CylinderVolume"
       "Area",         Area,
       "Description",  Concat(Name, ": Cylinder Volume"),
       "DeviceTag",    PollDriverName,
       "Address",      "F8:0",
       "ScanRate",     Invalid,
       "UnscaledMin",  0,
       "UnscaledMax",  100,
       "ScaledMin",    0,
       "ScaledMax",    100,
       "Units",        "gal",
       "AlarmLo",      Invalid,
       "AlarmHi",      Invalid,
       "PriorityLo",   Invalid,
       "PriorityHi",   Invalid,
       "InhibitLo",    1,
       "InhibitHi",    1,
       "AlarmSound",   Invalid,
       "ManualValue",  Invalid,
       "Threshold",    1,
       "Questionable", 0,
       "Quality",      Invalid,
       "DisplayOrder", 1,
       "HelpKey",      Invalid);
```

This creates a child analog status tag named SOMETHING\CylinderVolume.

Example 2 – Creating a persistent tag by supplying a series of name, value pairs.

```
Code\StartTag(Code\VTSDB, 0b11, "AnalogStatus",
       "Name",         "CylinderVolume"
       "Area",         Area,
       "Description",  "My Cylinder Volume"),
       "DeviceTag",    PollDriverName,
       "Address",      "F8:0",
       "ScanRate",     Invalid,
       "UnscaledMin",  0,
       "UnscaledMax",  100,
       "ScaledMin",    0,
       "ScaledMax",    100,
       "Units",        "gal",
       "AlarmLo",      Invalid,
       "AlarmHi",      Invalid,
       "PriorityLo",   Invalid,
       "PriorityHi",   Invalid,
       "InhibitLo",    1,
       "InhibitHi",    1,
       "AlarmSound",   Invalid,
```

```
        "ManualValue",  Invalid,
        "Threshold",    1,
        "Questionable", 0,
        "Quality",      Invalid,
        "DisplayOrder", 1,
        "HelpKey",      Invalid);
```

This creates a persistent analog status tag named CylinderVolume.

Example 3 – creating a child tag using a dictionary of parameters.

```
ParmsDict = Dictionary();
ParmsDict["Name"]        = "P1";
ParmsDict["Area"]        = "West";
ParmsDict["Description"] = "P1 West";
ParmsDict["IODevice"]    = "Mod1";
ParmsDict["Address"]     = 100;
Code\StartTag(ParentRoot, 0b11, "Parent", ParmsDict);
```

Example 4 – creating a child tag using an array of parameters:

```
ParmsArray = New(7);
ParmsArray[0] = "P2";
ParmsArray[1] = "East";
ParmsArray[2] = "P2 East";
ParmsArray[3] = "Mod1";
ParmsArray[4] = 200;
ParmsArray[5] = 2;
Code\StartTag(ParentRoot, 0b11, "Parent", ParmsArray);
```

Example 5 – Creating a child tag using parameters in an array of names and an array of values

```
ParmNames     = New(5);
ParmNames[0]  = "Area";
ParmNames[1]  = "Name";
ParmNames[2]  = "Description";
ParmNames[3]  = "Address";
ParmNames[4]  = "IODevice";
ParmValues    = New(5);
ParmValues[0] = "North";
ParmValues[1] = "P4";
ParmValues[2] = "P4 North";
ParmValues[3] = 400;
ParmValues[4] = "Mod1";
Code\StartTag(ParentRoot, 0b11, "Parent", ParmNames, ParmValues);
```

Example 6 – Creating a child tag using parameter Name/Value pairs (parms out of order)

```
Code\StartTag(Code\VTSDB, 0b11, "Parent",
              "IODevice",    "Mod3",
              "Area",        "Central",
              "Name",        "P5",
              "Description", "P5 Central",
```

```
                    "Address",      500,
                    "NotAParm",     0);
```

StateList

Description: Returns a list of states for a module.

Warning: This function should be used by advanced users only.

Returns: Array

Usage: Script Only.

Function Groups: Compilation and On-Line Modifications, State

Related to:

Format: StateList(Module, Option)

Parameters:

Module

Required. Any expression for the module or object.

Option

Required. Any numeric expression which indicates the data to list

Option	List Data
0	Name
1	Code value

StatementInstance

Description Takes a given code value and object and returns a code pointer value for that instance.

Warning This function should be used by advanced users only.

Returns Code Pointer

Usage Script Only.

Function Groups Compilation and On-Line Modifications, State

Related to:

Format: StatementInstance(Object, Statement)

Parameters

Object

Required. Any object expression.

Statement

Required. Any code value expression for the statement.

StateName

Description Returns the text name of the given state.

Warning This function should be used by advanced users only.

Returns Text

Usage Script Only.

Function Groups Compilation and On-Line Modifications, States

Related to:

Format: StateName(State)

Parameters

State

Required. Any code value expression for the state.

StaticSize

Description: If the variable provided in the parameter is static, this will return the size of that variable.

Returns: Numeric

Usage: Script Only.

Function Groups: Variable Functions

Related to:

Format: StaticSize(Var)

Parameters:

Var

Required. Any variable to test.

Comments: If the variable is not static, this function will return Invalid.

Examples:

```
varSize = StaticSize(myVar);
```

StatsWin

(ODBC Manager Library)

Description Called to display a window showing current ODBC driver statistics for both the main driver and any user-selected IO device addresses

Returns Nothing

Usage Steady State only.

Related to:

Format: \ODBCManager\StatsWin(Enable, X, Y, ConnectionName)

Parameters

Enable

Required. Set to true to display the statistics window

X

Required. The screen X position of the window

Y

Required. The screen Y position of the window

ConnectionName

Required. The name of the database connection to be

shown.

Com-
ments

Data Source	DSN
db State	dbState
Last Execution Time	ExecTime
Last Execution Date	ExecDate
Connection Time	Condt
Execution Time	Cmddt
Sem Q Time	Qdt
Average Execution Time	ACmddt
Average Sem Q Time	AQdt
Execution Count	ExecCount
Error Count	ErrCount
Recovery Count	RecoveryCount
db Open Count	OpenCount
db Close Count	CloseCount
Semaphore Count	SemCount
Error Code	Error
Error	Error
Error Time	ErrorTime
Error Date	ErrorDate
Error Query	ErrorQuery
ODBC Error Code	ODBCErrorCode
ODBC Error Current	ODBCErrorMsg
ODBC SQL State	ODBCSqlState
Last Error Code	LastError
Last Error	LastError
Last Error Query	LastErrorQuery

Step

Description: Transforms a continuous value into discrete steps.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Rounding Math

Related to: Ceil | Int | Scale

Format: Step(X, Size, Increment)

Parameters:

X

Required. Any numeric expression giving the value to reduce to discrete steps.

Size

Required. Any numeric expression giving the size of the steps in the X input parameter.

Increment

Required. Any numeric expression giving the size of the step in the result.

Comments: The return value is arrived at by dividing X by Size and taking only the portion before the decimal point, then multiplying by Increment. Note that negative numbers are taken down to the next lower number (e.g. -0.1 becomes -1). If any parameters are invalid, the return value is invalid.
This function is useful for determining the coordinates for menu highlight bars and for rounding numbers to the nearest multiple of another value.

Examples:

```
p = Step( 4, 1.5, 2);
q = Step( 2.31, 2, 10);
r = Step(10, 100, 0.25);
```

The values for p, q and r will be 4, 10 and 0 respectively.

Stop

Description: Causes the immediate termination of VTScada, closing all windows.

Returns: Nothing (optional return parameter may be used)

Usage: Script Only.

Function Groups: Software and Hardware

Related to: Slay | WindowClose |

Format: Stop([ExitValue])

Parameters:

ExitValue

An optional parameter that can be used to return a value to the calling program or batch file when VTScada exits. If not specified, the exit value from VTScada is 0, indicating "normal exit".

Comments: Not recommended in general, and especially not recommended for use in page code, where it may cause an untidy exit.
Shutdown resulting from the Stop() function may take a long time complete.

StrCmp

Description: Performs a case sensitive comparison of two text expressions and returns an indication of whether the first string is greater than, less than or equal to the second.

Returns: Numeric

Usage: Script or steady state.

Function Groups: String and Buffer

Related to: StrLen

Format: StrCmp(String1, String2)

Parameters:

String1

Required. Any text expression giving the first string to be used for the comparison.

String2

Required. Any text expression giving the second string to be used for the comparison.

Comments: When one string is referred to as greater than or less than another, it means each character has been converted to its ASCII value and the two strings have been compared character value by character value. In ASCII, the alphabet is numbered in ascending order, and uppercase letters are listed first, so their values are less than lower case letters. This function returns the following values:

Value	Meaning
-1	String1 is less than String2
0	String1 is equal to (identical to) String2
+1	String1 is greater than String2

Examples:

```
p = StrCmp("ABC", "abc");
q = StrCmp("AbC", "ABc");
r = StrCmp("ABC", "ABCD");
s = StrCmp("ABC", "ABC");
```

The values for p, q, r and s will be -1, 1, -1 and 0 respectively.

StreamEnd

Description: Returns whether or not a stream is at the end.

Returns: Boolean

Usage: Script Only.

Function Groups: Stream and Socket

Related to: BuffStream | ClientSocket | FileStream | GetStreamLength | PipeStream | ServerSocket | TCPIPReset

Format: StreamEnd(Stream)

Parameters:

Stream

Required. Any stream to test.

Comments: This function returns true (1) if the current stream pointer for Stream is at the end of the stream and false (0) otherwise.

Example:

```
If MatchKeys(2, "r");
[
  SRead(file1, "%d%n", val, filePos);
  end = StreamEnd(file1);
]
ZText(10, 30, Concat("At end of file ? ", end), 11, 0);
ZText(10, 40, Concat("Position in file = ", filePos), 11, 0);
ZText(10, 50, Concat("The value is ", val), 11, 0);
```

In this example, every time the user presses the letter r on the keyboard, a new data value is read and a check is done on whether or not the end of the file has been reached. All of these values are written to the screen. Note that the position in the file (filePos) that is displayed by the ZText command is the position after the read has taken place.

StrICmp

Description: Case Insensitive Text Comparison

Returns: Numeric

Usage: Script or steady state.

Function Groups: String and Buffer

Purpose: This function performs a case insensitive comparison of

two text expressions and returns an indication of whether the first string is greater than, less than or equal to the second.

Related to: StrCmp | StrLen

Format: StrICmp(String1, String2)

Parameters:

String1

Required. Any text expression giving the first string to be used for the comparison.

String2

Required. Any text expression giving the second string to be used for the comparison.

Comments: When one string is referred to as greater than or less than another, it means each character has been converted to its ASCII value and the two strings have been compared character value by character value. In ASCII, the alphabet is numbered in ascending order, and uppercase letters are listed first, so their values are less than lower case letters. This function returns the following values:

Return Value	Meaning
-1	String1 is less than String2
0	String1 is equal to (identical to) String2
+1	String1 is greater than String2

Lower case letters are considered to be the same as the corresponding upper case letters. For example, "VTS" and "vts" are considered to be identical.

Examples:

```
p = StrICmp("ABC", "AbC");
q = StrICmp("ABC", "ABCD");
r = StrICmp("ABCE", "abcde");
```

The values of p, q and r are 0, -1 and 1 respectively.

StrJustify

System Module

Description: Performs a word-wrap such that the string will break on or before each multiple of the maximum line length.

Returns: Text

Usage: Script Only.

Function Groups: String and Buffer

Related to: StrLen

Format: \System\StrJustify(Str [, Width])

Parameters:

Str

Required. The string to be formatted.

Width

An optional parameter indicating the maximum width for line length. If Width is Invalid, the width will be taken from the PrintWidth Setup.ini variable (see "Setup.ini [SYSTEM] Section Variables").

Comments: Inserts CR LF characters (0D 0A), either at each multiple of Width characters or after the first space (20) preceding Width characters.

Examples:

```
If Watch(1);
  [
    var1 = "AllDayLong";
    var2 = \system\StrJustify(var1, 6);
    var3 = "AllDay Long";
    var4 = \system\StrJustify(var3, 6);
    var5 = "All Day Long";
    var6 = \system\StrJustify(var5, 6);
  ]
```

After running, the byte codes of var1 through var 6 are:

```
var1:   41 6C 6C 44 61 79 4C 6F 6E 67
var2:   41 6C 6C 44 61 79 0D 0A 4C 6F 6E 67
```

```
var3:   41 6C 6C 44 61 79 20 4C 6F 6E 67
var4:   41 6C 6C 44 61 79 0D 0A 20 4C 6F 6E 67
var5:   41 6C 6C 20 44 61 79 20 4C 6F 6E 67
var6:   41 6C 6C 20 0D 0A 44 61 79 20 0D 0A 4C 6F 6E 67
```

StrLen

Description: Returns the length of a text string

Returns: Numeric

Usage: Script only.
May be used in optimized Tag Parameter Expressions.

Function Groups: String and Buffer

Related to: CharCount | Concat | Replace | SubStr

Format: StrLen(String)

Parameters:

String

Required. Any text expression.

Comments: This function does not return the number of characters in String, but rather, the number of bytes, including 0 (NULL) bytes. May be done on a stream value, but if the stream is longer than 65,523 characters, the return value will be 65,523.

Example:

```
v = StrLen("ABCDEF");
w = StrLen("ABC_DEF");
x = StrLen("ABC DEF");
```

The values of v, w and x are 6, 7 and 7 respectively.

Struct

Description: Creates a new dictionary and loads it with the keys listed in the parameters.

Returns: Dictionary structure

Usage: Script Only.

Function Groups: Variable Functions

Related to: Dictionary | MetaData

Format: Struct(A, B[, C ...]);

Parameters:

A

The first value to add to the dictionary.

B, C, etc.

Subsequent values to be added.

Comments: The dictionary created by this function is case insensitive and has a root value of NULL. The keys are assigned W_LONG values incrementing from zero that match their parameter positions. All of this is done with the idea that the resulting dictionary can then be used as a structure definition.

Examples:

```
S = Struct("A", "B");
```

SubStatementIndex

Description: Returns the index of a function within the statement where it is called.

Warning: This function should be used by advanced users only.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Compilation and On-Line Modifications, States

Related to:

Format: SubStatementIndex(CodePointer)

Parameters:

CodePointer

Required. Any code pointer value expression for the function.

Comments: This function returns 0 if CodePointer is a statement.

SubStr

Description: Returns a string that is a portion of another string.

Returns: Text

Usage: Script or steady state.

Function Groups: String and Buffer

Related to: Concat | StrLen | Replace

Format: SubStr(String, Start[, Length])

Parameters:

String

Required. Any text expression giving the original string to extract the sub-string from.

Start

Required. Any numeric expression giving the character offset from the start of String of where to start the resulting string. A value of 0 refers to the first character in String. If the Start is past the end of String, the result is invalid.

Length

Optional. Any numeric expression giving the number of characters to include in the resulting string starting with Start. Defaults to string length.

Comments: If any parameter of this function is invalid, the return value is invalid. If length exceeds the remaining number of characters in the string past Start, then only the remaining characters will be returned.

Examples:

```
piece1 = SubStr("ABCDEF", 2, 3);
piece2 = SubStr("ABCDEF", 3, 4);
```

The values of piece1 and piece2 are "CDE" and "DEF" respectively.

Sum

Description: Returns the arithmetic sum of all the valid array elements in a specified portion of a numeric array.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Array, Generic Math

Related to: AMin | AValid | FiltHigh | FiltLow | FitOffset | FitSlope | Mean | SDev | SumBuff | Variance

Format: Sum(ArrayElem, N)

Parameters:

ArrayElem

Required. Any array element giving the starting point in the array. The subscript for the array may be any numeric expression. If processing a multidimensional array, the usual rules apply to decide which dimension should be used.

N

Required. Any numeric expression giving the number of array elements to use starting at the element given by the first parameter.

Comments: Invalid array elements are not included in the calculation. The function returns an invalid result if either of its parameters is invalid, if there are no valid numerical array elements in the specified range, or if the number of elements to use is 0.

Example:

```
data[0] = 1;
data[1] = Invalid();
data[2] = 1.5;
data[3] = 3;
data[4] = 100;
dataSum = Sum(data[0], 4);
```

The value of dataSum is 5.5.

SumBuff

Description: Returns the summation of bytes in a buffer.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Generic Math, String and Buffer

Related to: BuffOrder | BuffRead | BuffStream | BuffToArray | BuffToParm | BuffToPointer | BuffWrite | Sum

Format: SumBuff(Buffer, Offset, N, Increment)

Parameters:

Buffer

Required. Any text expression giving the buffer to sum.

Offset

Required. Any numeric expression that gives the buffer offset from 0 to start the sum.

N

Required. Any numeric expression that gives the number of bytes to sum. If N is negative, the absolute value of N is used, but the operation is changed from summation to XOR (each successive byte is XORed with rather than added to the first byte).

Increment

Required. Any numeric expression that gives the incre-

mental step of the sum in bytes.

Comments: This function returns the 32 bit sum of N bytes in Buffer, starting at Offset and stepping by Increment bytes. This function is useful for computing checksums for serial communications.

Examples:

Given that myBuff is length 256, and values are the bytes from 0 to 255:

```
sum1 = SumBuff(myBuff, 0, 256, 1);
```

sum1 would add 0 + 1 + 2 + ... + 254 + 255

```
sum2 = SumBuff(myBuff, 5, 100, 1);
```

sum2 would add 5 + 6 + 7 + ... + 103 + 104

```
sum3 = SumBuff(myBuff, 1, 100, 2)
```

sum3 would add 1 + 3 + 5 + ... + 197 + 199

To compute a 16 bit checksum which adds 16 bit words in a 100 byte buffer with the low byte first:

```
serCheckSum = And(SumBuff(buff2, 0, 50, 2) +
256 * SumBuff(Buffer, 1, 50, 2), 0xFFFF);
```

SWrite

Description: Performs a formatted write of ASCII or binary data to a pre-existing stream and returns the number of data items not written.

Returns: Numeric

Usage: Script Only.

Function Groups: Stream and Socket

Related to: BuffWrite | FileSize | Format | FWrite | GetStreamLength | Print | PrintLine | Redirect | Save | SRead | StreamEnd

Format: SWrite(Stream, Format, V1, V2, ...)

Parameters:

Stream

Required. A stream as returned from BuffStream or FileStream.

Format

Required. Any text expression giving the format of how the values (Vn parameters) are to be written. This format is similar, but not identical, to the C language format string for the printf function, whereby each of the Vn parameters in the statement is assigned to a % format specification in the order in which each appears in the list.

Note that like a standard text string, these format specifiers must also be enclosed by double quotes.

If a format specification appears for which there are no remaining V parameters, the format specification characters themselves are output to the stream exactly as they appear in the Format.

For the % format specifications, the following form applies (where the [ ] indicates optional elements):

%[-][+][SPACE][#][width][.precision]type

where

% (percent sign) is mandatory;

- (minus sign) causes the data to be left justified within the field (for binary types b and ASCII character types c, this option is ignored);

+ (plus sign) causes positive numbers to be prefaced with a + sign (negative numbers are unaffected). This allows easy alignment of positive and negative numbers in a printed column of numbers. For binary types b and non-numerical

types, this option is ignored;
space represents the single space character, and is similar to the [+] option but places a single space rather than a plus sign in front of positive numbers (negative numbers are still unaffected). This allows alignment of a column of numbers without having to show all signs. For binary types b and non-numerical types, this option is ignored;
(hash mark) When used with the o , x , or X format, the # flag prefixes any nonzero output value with 0, 0x, or 0X, respectively.
width is a number that specifies the minimum number of characters to output. Numbers that require more characters than specified by the width value are truncated on output. If the number of characters in the number or string is less than width, blanks will be added to the left or right, depending upon whether the output is left or right justified (i.e. whether or not the [-] option has been specified) until the width is reached. For binary types b and ASCII character types c, this option is ignored;
precision has a different meaning for each of the type options as follows:

- Integer types d, l, u, o, x, and X precision specifies the minimum number of digits to output. If the number contains fewer digits, leading zeroes will be added to the left of the number. If precision is 0, omitted, or if the decimal point appears without a number following it, the precision defaults to 1. The number is not truncated.

- Floating point types e and E precision specifies the number of digits after the decimal point. The last digit is rounded. The default precision in this case is 6. If the precision is 0 or if the decimal point appears without a number following it, no decimal point appears in the output.
- Floating point type f precision specifies the number of digits after the decimal point. The last digit is rounded. The default precision is 0. If the precision is explicitly 0, no decimal point is output. If a decimal point is output, at least one digit will be placed before the decimal point.
- Floating point types g and G precision specifies the maximum number of significant digits to be output. If no precision is specified, all significant digits are written.
- String type s precision specifies the maximum number of characters of the string to be output. If the string contains more characters than specified by the precision, the string is truncated and only the first characters are written. If the precision is not specified, all of the string characters are output.
- ASCII character type c The precision option is ignored.
- Binary type b The precision option is ignored.

x unsigned hex integer using "abcdef"
znnn Escape character where nnn is the 3-digit ASCII code
type is mandatory. The type specification must be one of those listed below.
Note: The case of the letter is important. Specifying a character for the type that is not in this list will result in all the characters following the

% up to that point to be output exactly as they appear in the Format string.

Type	Meaning
nb	Binary format, where n is a number indicating the type of value (see below).
c	Single ASCII character (byte)
d	Signed decimal integer
e	Signed exponential; exponent key is "e".
E	Signed exponential; exponent key is "E".
f	Signed floating point.
g	e or f format, whichever is shorter.
G	E or f format, whichever is shorter.
h	Handle to a window.
i	Signed decimal integer.
o	Unsigned octal integer.
p	Pointer to a buffer.
s	Text string.
u	Unsigned decimal integer.
x	Unsigned hex integer using "abcdef".
X	Unsigned hex integer using "ABCDEF".

nb, Binary type For the format specification of %nb, where n specifies the type of number, n must be a single digit from one of the following choices. All are low-byte-first.

n	Value Type
0	Byte, unsigned

1	Signed short integer (2 bytes)
2	Signed long integer (4 bytes)
3	IEEE single precision float (4 bytes)
4	<obsolete>
5	IEEE double precision float (8 bytes)
6	<obsolete>
7	Unsigned short integer (2 bytes)
8	Unsigned long integer (4 bytes)

Note: Other options such as width and precision do not apply to the b type.

c, ASCII character type: This type is not representative of a single character in a string, but rather, represents single byte ASCII characters. Input values (the Vn parameter to which this format specification applies) must be integers in the range of 0 to 255 in order for the output to be a valid ASCII equivalent character. Strings are not acceptable input values. Note that the %c format specifier behaves differently when used in an output statement such as BuffWrite than when used in an input statement, such as BuffRead.

d, Signed decimal integer:

e, Signed exponential: Exponent key is “e”

E, Signed exponential: Exponent key is “E”

f, Signed floating point

g, e or f formats: Whichever is shorter

G, E or F formats: Whichever is shorter

h, Window handle type: This type is used for building structures to be handed to DLLs and

should be used by advanced users only.

p, Buffer pointer type: This type is also used for building structures to be handed to DLLs and should be used by advanced users only.

s, Text string type:

Plain text Text in the Format parameter is written exactly as it appears, with three exceptions:

- Percentage sign (%) Since format specifications for the Vn parameters are indicated by a percentage sign, to include an actual percentage sign as part of the Format parameter, precede it with a backslash character (i.e. \%).
- Backslash character (\) Since this is used to indicate special control characters such as line feed, carriage return, and form feed, to write a backslash as part of the Format parameter, use two backslash characters (i.e. \\).
- Quotation marks (") The entire test string is delimited by quotation marks, so to include a set of quotation marks as part of the Format parameter, use a set of quotations marks (i.e. "").

Control characters In order to encode certain control characters as part of the Format parameter, one of two methods may be used. The first is to use a backslash character followed by one of the single character codes listed below to produce the desired result (notice that the letters must be lower case):

Code	Meaning
\b	Backspace
\f	Form feed
\n	Line feed

\r	Carriage return
\t	Horizontal tab
\v	Vertical tab

\nnn In addition to the above predefined codes, \nnn may be used, where nnn is a three digit integer in the range of 0 to 255 specifying a certain ASCII character. If the number contains less than three digits, the leading spaces must be padded with zeroes; this is not the case with the previously listed single character control characters. For example, to include the one byte ASCII character G in the output, you could place its decimal equivalent of 71 in the Format string as \071.

u, Unsigned decimal integer,

x, Unsigned hex integer using "abcdef"

X, Unsigned hex integer using "ABCDEF"

Offset is any numeric expression giving the starting buffer position in characters or bytes for the write, starting at 0.

V1, V2, ...

Required. Any expressions giving the values to be output in the form described by the Format parameter. Each of the Vn parameters is evaluated and written in the order in which each appears in the parameter list. The way in which they are formatted is dictated by the % format specifications. V1 is formatted by the first % sequence in the Format parameter, V2 by the second, and so on. If there are more V parameters than % sequences in the Format string, the remainder are ignored. If there are fewer V parameters than %

sequences in the Format string, the remaining % sequences are written literally without any translation.

Comments:

You cannot write to a read-only file. You may use GetFileAttribs and SetFileAttribs to get/set the read-only attribute.
If one of the values to be written is outside of the range of the type indicated by the format specifier, a "0" is written.
If the value to be written is invalid, nothing is written for most format specifiers, with the exception of %nb, which will write a "0" in the place of the invalid. Please note that an invalid output does not prevent the execution of the SWrite function.
This function returns the number of Vn parameters not written to the stream. A 0 return value indicates success. Variables that contain invalid values that were not written due to their invalidity do not increment this count. An invalid return value indicates an error with one of the parameters.

Examples:

```
If MatchKeys(2, "W");
[
  b = MakeBuff(16, 65) { Create buffer; fill it with A's },
  buff = BuffStream(b) { Create the stream },
  SWrite(buff { Write to the stream },
        "A=%d B=%4.2d C=%5.2f\r\n%s\r\n" { Format },
        1, 2, 2/3, "Hello World" { Output values });
]
```

This statement would write the following two lines of text to the stream buff :

```
A=1 B= 02 C= 0.67
Hello World
```

If you replaced the third line of the script with the following:

```
SWrite(buff { Write to the stream },
       "\072\105\n%c%c%c" { Format },
       77, 111, 109 { Output values });
```

The following two lines would instead be written to buffer stream buff:

```
Hi
Mom
```

SystemSelf

Description Returns the object value of the system module for the given application.

Returns Object value

Usage Script or steady state.

Function Groups Basic Module

Related to: Self

Format: SystemSelf(Object)

Parameters

Object

Required. Any object value expression for any module instance in the application.

Comments There is one and only one instance of the system module for each application.

T Functions

The sections that follow identify all VTScada functions beginning with "T".

TableSynch

(ODBC Manager Library)

Description: Synchronizes the fields matching a specified criteria within matching tables in two databases. Should be run as a called module, waiting for completion. Do not call as a sub-routine.

Returns: Nothing

Usage: Script Only.

Function Groups: ODBC

Related to:

Format: \ODBCManager\TableSynch(DSNSource, DSNDest, TableName, WhereFields, WhereOperators, WhereValues, WhereSQLDataTypes, WhereAND, SourceUsername, SourcePass, DestUsername, DestPass, [TransObj,] NRecords, CurrRecord)

Parameters:

DSNSource

Required. Data source name of the database to retrieve data from.

DSNDest

Required. Data source name of the destination database to send data to

TableName

Required. Table name to read/write in both databases

WhereFields

Required. A text array of field names to select using an SQL WHERE clause

WhereOperators

Required. A text array of operators to use when selecting fields using an SQL WHERE clause

WhereValues

Required. A text array of values to use when selecting the above fields using an SQL WHERE clause

WhereSQLDataTypes

Required. Values indicating the data type of the Where values. Should be a simple value or an array matching the WhereFields parameter. Refer to Data Type Codes used in the ODBC Manager for a list of the codes.

WhereAND

Required. Set to true (1) if the fields in the SQL WHERE clause are to be AND'ed. Otherwise, OD'ed

SourceUsername

Required. User name for source db

SourcePass

Required. A password for source db

DestUsername

Required. A user name for destination db

DestPass

Required. A password for destination db

TransObj

The object value of transaction

NRecords

Required. A number of records to synchronize

CurrRecord

Required. Current record being written

Comments: This module is a member of the ODBCManager Library, and must therefore be prefaced by \ODBCManager\, as shown in "Format" above.

Tag

Description: Returns a Tag value, which works like (and in place of) a Normalize value.

Returns: Tag

Usage: Steady State only.

Function Groups: Graphics, Generic Math, Variable

Related to: Normalize | Rotate | Scale | Trajectory

Format: Tag(Value, LowInput, HighInput, LowScale, HighScale, Mode, Freq)

Parameters:

Value

Required. Any numeric expression to be normalized.

LowInput

Required. Any numeric expression, which represents the lowest normal input value of Value. This is not a limit.

HighInput

Required. Any numeric expression, which represents the highest normal input value of Value. This is not a limit.

LowScale

Required. Any numeric expression, which represents the lowest normal scaled value of Value. This is not a limit.

HighScale

Required. Any numeric expression, which represents the highest normal scaled value of Value. This is not a limit.

Mode

Required. Any numeric expression that specifies how this I/O tag is to be simulated. If the application isn't simulating I/O, this parameter is ignored, but must be present and valid.

Freq

Required. Any numeric expression that specifies the frequency at which the simulation runs. This applies only if this tag is simulated, and only if the Mode specifies a cyclic simulation; however, this parameter must always be present and valid.

Comments: This function scales an expression from the low and high input range to between low and high values. The return

value is a Normalize value.
If a Tag value is used in an expression, it will return the scaled value.

Example:

```
pumpFlow = Tag(rawPumpFlow, 0, 4095, 0, 150);
```

This sets the variable pumpFlow to a Tag value. The variable rawPumpFlow is scaled from between 0 to 4095, to between 0 and 150. This Tag value might be used to scale a rectangle (to show a bar graph of the pump flow), as shown:

```
GUIRectangle(0, 100, 100, 0 { Rectangle bounding box },
             1, 1, 1, pumpFlow, 1 { Scale top of bar only },
             0, 0 { No trajectory, rotation },
             1, 0 { Visible; reserved },
             0, 0, 0 { No focus, selection },
             12, 15 { Lt red, white outline });
```

The variable pumpFlow could also be reused in other expressions, graphics functions, rotations, or trajectories.

TagIconMarker

Description: Draws an "IconMarker" in the center of its transform area, and optionally shows a blank icon when in editing mode.

Returns: Object

Usage: Steady State only.

Function Groups: Graphics

Related to:

Format: TagIconMarker(TagReference[, ShowInConfigParm])

Parameters:

TagReference

Required. A reference to the tag for which the unlinked widget, questionable, manual or dummy tag indicator should be shown.

ShowInConfigParm

Optional Boolean. If TRUE, the marker will be visible (but blank) in the Idea Studio. Defaults to FALSE.

Comments:

Examples:

```
GUITransform(0, 30, 30, 0,
             1, 1, 1, 1, 1 { Scaling },
             0, 0 { Movement },
             1, 0 { Visibility, Reserved },
             0, 0, 0 { Selectability },
             Variable("Code\Library")\TagIconMarker(\Root));
```

Tan

Description: Returns the trigonometric tangent of an angle in radians.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Trigonometric Math

Related to: ACos | ASin | ATan | Cos | Sin

Format: Tan(Angle)

Parameters:

Angle

Required. Any numeric expression giving the angle in radians.

Comments: The returned value is a number in the range of -1.00 to +1.00. To convert an angle from degrees to radians multiply by \pi / 180 or (approximately) 0.0174533.

Example:

```
x = Tan(45 * \pi / 180);
```

The value of x will be 1.

Target

Description: Returns an indication of whether the locator (e.g. mouse) is within a specified screen area.

Returns: Boolean

Usage: Script or steady state.

Function Groups: Locator

Related to: Click | Pick | SetXLoc | SetYLoc | XLoc | YLoc | GUITransform

Format: Target(X1, Y1, X2, Y2)

Parameters:

X1

Required. Any numeric expression giving the X coordinate on the screen of one side of the screen area ("target").

Y1

Required. Any numeric expression giving the Y coordinate on the screen of either the top or bottom of the screen area ("target").

X2

Required. Any numeric expression giving the X coordinate on the screen of the side of the "target" opposite to X1.

Y2

Required. Any numeric expression giving the Y coordinate on the screen of either the top or bottom of the target, whichever is the opposite of Y1.

Comments: This function returns true if the locator position is within the boundaries of the "target" ((X1,Y1) – (X2,Y2)). If the locator is not installed, the function will return false (0).

Note: This function is disabled when using a GUITransform as a GUIStrectch.

Example:

```
If Target(120, 50, 220, 80);
[
   ...
]
```

This statement will cause the script to execute whenever the mouse passes over the target area.

TCPIPReset

Description: Shuts down and resets all TCP/IP functions.

Returns: Nothing

Usage: Script Only.

Function Groups: Network

Related to: ClientSocket

Format: TCPIPReset()

Parameters: None

Comments: This statement may only appear in a script.

TempFileStream

Description: Creates a temporary file on disk and connects a stream to the temporary file. The temporary file is removed when the stream is closed or no longer referenced or if the VTScada process is terminated.

Returns: Stream

Usage: Script or steady state.

Function Groups: File I/O, Stream and Socket

Related to: BlockWrite | BuffStream | ClientSocket | CloseStream |

FileStream | GetStreamLength | PipeStream | SRead | StreamEnd | SWrite

Format: TempFileStream(Buffer)

Parameters:

Buffer

Required. Any text or buffer expression. This serves as the initial content of the stream.

Comments: This function returns a stream connected to a temporary disk file with the contents of Buffer; the pointer at which an action (read or write) begins will be at the start of the stream. Writing to this stream can overwrite or expand (or both,) the size of this initial stream and file.

Example:

```
Stream = TempFileStream("0123456789");
```

The variable stream would contain "0123456789".

Text

Note: Deprecated. Do not use in new code.

Description Displays text in a window.

Returns Nothing

Usage Steady State only.

Function Groups Graphics

Related to: GUIText | Output | TextAttribs | ZText

Format: Text(X, Y, Value, Foreground, Fill, Background, Size, Obsolete)

Parameters

X

Required. Any numeric expression giving the X screen

coordinate of the lower left corner of the text on the screen.

Y

Required. Any numeric expression giving the Y screen coordinate of the lower left corner of the text on the screen.

Value

Required. Any text expression giving the value to be displayed.

Foreground

Required. Any numeric expression giving the color of the characters to be displayed.

Fill

Obsolete - set to zero.

Background

Required. Any numeric expression giving the color of the background area for the output characters.

Size

Required. Any numeric expression giving the height of the characters in units of Y screen coordinates. If this value results in a specification of less than 12 screen pixels high, the text will be the small text (8 pixels high); otherwise, the text will be the large text. If Size is negative, it will be interpreted as a dot text output of size equivalent to the absolute value of the size. The number will be displayed to the nearest multiple of the base 8 pixel by 8 pixel text. This produces faster, non-destructive large characters than the normal large text characters.

Setting size greater than 12 results in non-XOR drawing.

Obsolete

No longer used, but is maintained for backward compatibility with previous versions of VTS; set to 0.

Comments

This statement has been superseded by the GUIText and ZText statements and is maintained for backwards compatibility only.
As of version 11, this is now drawn in the same z-order as other graphics, making it similar to the z-graphics functions.

Note: Within an Anywhere Client session, this function does nothing.

TextAttribs

Description: Returns graphic-related information about a text, given a font.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Graphics

Related to: GUIText | Output | ZText

Format: TextAttribs(Text, Font, Option)

Parameters:

Text

Required. Any text expression.

Font

Required. Any font expression.

Option

Required. Any numeric expression for the desired para-

Option	Parameter
0	Width of text in user coordinates
1	Height of text in user coordinates

Example:

```
width = TextAttribs("Testing" { Text to display },
                    Font("ARIAL", 0, 16, 0, 5, 0, 0) { Font },
                    0 { Width in user coordinates });
```

The value of width will be 61, which is the width of the given text string in the given font.

TextBox

(System Library)

Description: Displays a text string, breaking it into multiple lines at space or CRLF.

Returns: Nothing

Usage: Steady State only.

Function Groups: Graphics

Related to:

Format: \System\TextBox(X1, Y1, X2, Y2, Msg, FontVal, Style, BevelTitle [,BackColor, BodyTextColor, HorizontalAlign, VerticalAlign, Height])

Parameters:

X1

Required. The left side coordinate.

Y1

Required. The bottom side coordinate.

X2

Required. The right side coordinate.

Y2

Required. The top side coordinate.

Msg

Required. The message buffer to display.

FontVal

Required. The font to use (defaults to _DialogFont).

Style

Required. The bevel/border/wordwrap requirements. This can be one of:

Bit 0	Draw a bevel at X1/Y1/X2/Y2. An 8-pixel margin will be added when determining the text rectangle. The default is TRUE.
Bit 1	Draw a border around the text rectangle. A further 4-pixel margin will be added. The default is TRUE.
Bit 2	Word wrap text. The default is TRUE.
Bit 3	Transparent background. Can be clicked-through to the controls beneath. Default is FALSE
Bit 4	Disables the vertical scroll bar when TRUE. Default is FALSE
Bit 5	Disables the horizontal scroll bar when TRUE. Default is FALSE.

BevelTitle

Required. The title to display for the bevel (if relevant).

BackColor

An optional parameter specifying the background color Defaults to #SYS…BUTTONFACE.

BodyTextColor

An optional parameter specifying the text color. Defaults to 0, black.

HorizontalAlign

An optional numeric value, specifying the horizontal alignment of the text.
0 == Left,

1 == Center,
2 == Right.
Defaults to 1, Centered.

VerticalAlign

An optional numeric value, specifying the vertical alignment of the text.
0 == Top,
1 == Center,
2 == Bottom.
Defaults to 1, Centered.

Height

Optional Numeric. The height of the text, not including borders and bevels will be *returned* to the caller in this parameter.

Comments: If the current text cannot be accommodated by breaking at a word boundary, then the word will be hyphenated (not grammatically). A vertical scrollbar will be automatically added if required.

TextIP2Bin

(RPC Manager Library)

Description: Returns the Binary representation of the specified IP.

Returns: Text

Usage: Script Only.

Function Groups: String and Buffer, Network

Related to: BinIP2Text

Format: \RPCManager\TextIP2Bin(IP)

Parameters:

IP

Required. The IP that you want converted to Binary format. Formatting example 192.168.0.200.

Comments: This subroutine is a member of the RPC Manager's Library, and must therefore be prefaced by \RPCManager\, as shown in the "Format" section. If the application you are developing is a script application, the subroutine call must be prefaced by System\RPCManager\, and the System variable must be declared in AppRoot.src.
If you have an IP of the format 192.168.0.200/24, the /24 will be ignored.

TextOffset

Description: Returns the character offset to the definition text of a desired item.

Warning: This function should be used by advanced users only.

Returns: Numeric

Usage: Script Only.

Related to: AdjustCode | GetModuleText | GetOneParmText | GetParmText | GetStateText | GetTransitText | GetVariableText | SetModuleText | SetOneParmText | SetParmText | SetStateText | SetTransitText | SetVariableText | TextSize

Format: TextOffset(CodeValue, Type)

Parameters:

CodeValue

Required. Any code value expression for the item.

Type

Required. Any numeric expression for the value type of CodeValue.

TextSearch

Description: Returns the array index of the first occurrence of the given text key in an alphabetically ordered array.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Array, String and Buffer

Related to: ArrayStart | ArraySize | LookUp

Format: TextSearch(ArrayElem, N, Text, Case)

Parameters:

ArrayElem

Required. Any array element giving the starting index for the array operation. The index for the array may be any numeric expression.
If processing a multidimensional array, the usual rules apply to decide which dimension should be used.

N

Required. Any numeric expression for the number of array elements to search.

Text

Required. Any text expression to search for.

Case

Required. Any logical expression. If true, the search is case-sensitive; otherwise, the search is case-insensitive.

Comments: If the key is not found in the array, the function returns invalid. Notice that the array (or the elements being searched in the array) must be in ascending alphabetical order for this statement to return a valid value.

Example:

```
index = TextSearch(dataArray[0] { Start of search in array },
                   ArraySize(dataArray, 0)
                   { Search all elements },
                   "green" { Text string to search for },
                   0 { Case insensitive search });
```

Given that dataArray is an array of text strings of various color names, index will be set to the subscript of the array element whose entry is "green".

TextSize

Description: Returns the size in characters of the definition text of a desired item.

Warning: This function should be used by advanced users only.

Returns: Numeric

Usage: Script Only.

Function Groups: String and Buffer

Related to: AdjustCode | GetModuleText | GetOneParmText | GetParmText | GetStateText | GetTransitText | GetVariableText | SetModuleText | SetOneParmText | SetParmText | SetStateText | SetTransitText | SetVariableText | TextOffset

Format: TextSize(CodeValue, Type)

Parameters:

CodeValue

Required. Any code value expression for the item.

Type

Required. Any numeric expression for the value type of CodeValue.

Comments: This function may only appear in a script.

TGet

Note: Deprecated. Do not use in new code.

Description: This threaded function reads an array of historical data from a file (written by Save or SaveHistory) and returns an indication of parameter errors.

Returns: Numeric

Usage: Script Only.

Function Groups: Log, File I/O

Threaded: Yes

Related to: TGet | HistorianDeleteRecords | HistorianGetData | HistorianGetInfo | HistorianReadRecords | HistorianWriteRecords | Get | GetHistory | GetLogInfo | Save | SaveHistory

Format: TGet(Array, N, File, FieldNum, StartDate, StartTime, TPP, Mode [, ErrorCode, StaleTime, PathPrefix])

Parameters:

Array

Required. A variable which will be set to an array upon completion of the data retrieval.

N

Required. Any numeric expression giving the number of array entries to create.

File

Required. Any text expression giving the file name for the historical data file. The file extension should not be added to the name since the default of ".DAT" is automatically added.If the file name is prefixed with a period, the path will be to the directory the module is contained in.

FieldNum

Required. Any numeric expression giving the field number to be read from the file. The value number for the actual data starts at 0 and corresponds to the columns specified in Save or SaveHistory. It is also possible to retrieve time data associated with each record by setting this parameter to a negative value.Time options are:

FileNum	Time Option
-1	Time of day only
-2	Date only
-3	Time since January 1, 1970

It is possible to retrieve more than one field in a single TGet statement. To do this, pass an array of values in as the FieldNum parameter.

StartDate

Required. Any numeric expression giving the date to search for in the file as the starting date for the data. This date is expressed as the number of days since January 1, 1970. If StartDate is less than zero, it indicates the relative file position to read rather than the date.

A -1 indicates the last entry in the file. A -2 the second last and so on.

It is important to note that in cases where Save has been used with a non-negative Buffers parameter, the last entry in the file will be invalid (a fact that is useful in data logging). In cases such as this, the first valid entry will be indicated by -2, the second by -3 and so on.

StartTime

Required. Any numeric expression giving the time of day on StartDate to search for in the file as the starting

time for the data. This time is expressed as the number of seconds since midnight on StartDate.

It is legal for StartTime to be greater than one day. It is legal for StartTime to be negative, where data will start the previous day at (86400 - StartTime) seconds after midnight. If StartDate is less than zero, StartTime is ignored.

TPP

Required. Any numeric expression giving the time span in seconds for each array entry. Each array element will contain the data which correspond exactly to this time period which corresponds to 0 or more data points in the file. If TPP is positive and FieldNum selects a text value, the first entry which falls in a time is read, and Mode is ignored.

If TPP is equal to 0, the data is read from the file and placed in the array on a one to one basis.

If TPP is less than 0, the data is read on a one to one basis from the StartDate and StartTime for (negative) TPP seconds, TPP places an upper limit on the time span to read. In both of these cases, the Mode parameter is ignored.

Mode

Required. Any numeric expression giving the method of handling the data. If TPP is greater than 0, the values that fall in each time span will be represented as follows:

Mode	Time Span Representation
0	Time weighted average
1	Minimum in range
2	Maximum in range
3	Change in value over the range
4	Value at start of range
5	Time of minimum in range (in seconds since Jan 1, 1970)
6	Time of maximum in range (in seconds since Jan 1, 1970)
7	Count the total number of zero to non-zero transitions within each TPP period.
8	Totalizes, for each TPP, the amount of time when the value is non-zero (Invalid is counted as zero).
9	Totalizes, for each TPP, the arithmetic sum of the recorded values.
10	Interpolates between values.

In the case of modes 5 and 6, FieldNum should still be set to indicate the field number on which the mode is to act; the return values will be times indicating the maximum or minimum in that field for each time span.

If TPP is less than or equal to 0, Mode is ignored. If the data is text, the first entry in a given time range is used for the array entry and

Mode is ignored.

It is possible to retrieve more than one mode in a single TGet statement. To do this, pass an array of values in as the Mode parameter.

ErrorCode

An optional parameter that is any variable which will be set to a valid value upon completion of the TGet. Its meaning is as follows:

Value	Error
0	No error
1	Parameter values out of described range
2	File could not be opened
3	Corrupted .DAT file
4	Field requested could not be found

If ErrorCode is not required but either StaleTime or PathPrefix is, then ErrorCode should be given as an Invalid value.

StaleTime

An optional parameter that sets a maximum validity duration for data elements that are being TPP processed. Normally, every data point is treated as remaining valid until the next data point. If a valid StaleTime parameter is given, then any data point will be treated as invalid StaleTime seconds after the recorded time. If TPP is less than or equal to 0, StaleTime is ignored. If StaleTime is not required but PathPrefix is, then StaleTime should be given as either an Invalid value or a constant zero.

It is possible to specify more than one stale time in a single TGet statement. To do this, pass an array of values in as the StaleTime parameter.

PathPrefix

An optional text expression parameter that enables and controls the retrieval of data from across a set of files. No default.

Comments:

This function is similar to Get except that it runs in its own thread - it is typically used when a large amount of data is being read. When the TGet is executed in its script, it starts its own thread and VTScada will continue executing; when it is finished executing, it will set the data in Array. Note that Array will not be initially invalidated upon execution of this statement. This means that if Array already contained data when the TGet was executed, that data will remain untouched until all of the data requested by TGet has been amassed, at which time Array will be set to its new value.

There is a return value for this function that indicates if any of its parameters are invalid. The function will immediately return a value of false (0) unless a parameter was invalid, in which case it will return true (1). Note that the return value only signals completion of the function's execution if it is true, otherwise the function will continue executing in its thread.

If StartDate is given a negative value, indicating that a particular entry is to be retrieved, it must be stressed that the file being read by the TGet may or may not contain an invalid record at the end of the file. If the Save that created the file was given a negative number for its Buffers parameter, the invalid record would not have been written to the file, however, a zero or positive value for Buffers will mean that the last record of the file will be one whose fields are all invalid and whose time and date stamp reflect the cessation of writing to the file by the Save.

If FieldNum is an array with more than one element, then TGet will retrieve multiple fields from the file.

In this case, ArrayElem must represent an array with at least two dimensions. The requested values will be returned in a manner analogous to GetHistory; that is, with the data for a column in the rightmost dimension, and the column index in the previous dimension.
When FieldNum is specified as an array, Mode or StaleTime, or both, may be specified as either a single value or an array of values. If a single value is specified, that value will be used for each of the fields specified in FieldNum. If an array of values is specified, the first element in the array will be applied to the first element of FieldNum, and so on.
If PathPrefix is specified, then this changes the interpretation of the File parameter. In this case, the referenced file is not the source of the data, but a file containing references to other files which are the data sources. This file should be in standard VTScada logfile format and should contain a file reference as the first text value of each record (other values are ignored). The records should be in the correct time order with respect to the data files. The value of the PathPrefix is a string, which when prefixed to one of the file references, will yield a full pathname to the target file. If no prefix is required, but expansion of the dataset is required, then PathPrefix should be an empty string.
If a filename entry does NOT begin with a "\" or "<drive letter>:\", then the PathPrefix will be prepended to the filename.
If a filename entry DOES begin with "<drive letter>:\", then the PathPrefix will NOT be prepended to the filename.

If a filename entry does begin with a "\", then the "<drive letter>:" from the PathPrefix will be prepended to the filename. If there is no "<drive letter>:" in the PathPrefix then the "<drive letter>:" from the path of the File parameter will be used instead.
PathPrefix would normally be the application path.

Example:

```
Init [
  If 1 Main;
  [
    TGet(trend { Destination array },
     100 { Get 100 array elements fr file },
        "G:\mix\mixtrend" { path and file name },
        0 { Read first 'column' from file },
        Today() { Starting today },
        Seconds() - 3600 { Starting 1 hour ago },
        18 { Each element covers 18 secs },
        0 { Time weighted average });
  ]
]
Main [
  If Valid(trend) DoneReading;
  ZText(10, 20, "Reading Data", 10, 0);
]
```

This reads a half hour of data from a file, starting one hour ago. Note that just after midnight, the expression Seconds( ) - 3600 may be negative; the TGet statement interprets this as before midnight on the previous day (which is correct). X is set to the record following the last record read from the file. Note also that a full path name may be specified, including network drives. Also note that it is irrelevant when data were logged to the file; the Save statement trigger could have been a regular timer (such as AbsTime), or an event (such as Change(level, 1)).

Thread

Description: Launches a module in its own separate thread.

Returns: The object value of the instance of the module that is launched into the new thread.

Usage: Script Only.

Function Groups: Basic Module

Related to: Call | FindVariable | Launch | LoadModule | ThreadHistory | ThreadList | ThreadName

Format: Thread(Module, Parent, Caller, Name, P1, P2, ...)

Parameters:

Module

Required. A module pointer or a text expression giving the module to run. If the module is in scope, the text expression giving its name may be used, otherwise the module pointer returned from either a LoadModule or FindVariable statement is typically used.

Parent

Required. The object value of the module where Module is to resolve its global variable references. If a valid non-object value is supplied Module will resolve its global variable references to the scope defined by the first parameter. If this is invalid, the module will still run, but global references will be invalid.

Caller

Required. The object value of the window to draw in. This specifies the module instance where Module acts as if it were called from. If this is invalid, the module will still run but will not stop without a Slay. If it is valid, the module will stop when the Caller module instance stops or when a Slay is executed upon it.

Name

Required. Any text expression giving the name that is to be associated with this particular thread. This is the name that will be returned by the ThreadList function.

P1, P2, ...

Required. Are any expressions which will be supplied as parameters to the module.

Comments: This function behaves similar to Launch except that a separate thread is created in which the module is executed. This means that it will not block execution of other modules; the CPU time will be divided equally amongst threads of equal priority. Great care should be exercised when using this function since each thread created by Windows™ uses certain system resources and will by its very existence slightly slow the running of the application. In general, the Thread function should only be used when a module would otherwise monopolize system resources to such an extent that other critical modules would be severely hampered in their execution.

This function returns an object value of the newly started module, in the same way that Launch does. This means that parameters are passed to the module as a value only, and if the module instance changes one of their values, its value will not change outside of the scope of the threaded module. Variables external to the module that the module itself will be required to alter should reside within the scope of Parent and be set directly, rather than passed as parameters.

Example:

```
If ! Valid(ptr);
[
  ptr = Thread("CheckTank" { Module to launch },
              Self(), Self() { Parent and caller },
                CheckTankThread { Name },
               pressure1, pressure2, level { Parameters });
]
```

ThreadHistory

Description: Returns in an array the history of execution for a specified

thread.

Returns: Array of text

Usage: Script Only.

Function Groups: Basic Module

Related to: Thread | ThreadList | ThreadName |

Format: ThreadHistory(ThreadName)

Parameters:

ThreadName

Required. The name of the thread for which you wish the history of execution to be returned.

Comments: The return value for ThreadHistory is a listing of the last 8192 operations on the thread (max) with five data points per operation. This array should be interpreted in the format, [column][row].
The five rows (data operations) are as follows:
0 = Time of execution
1 = Module, State and Statement (type 13) for the operation
2 = State name where the operation is located
3 = Statement number of the operation
4 = Object value for the owning module instance

ThreadIdle

Description: Returns TRUE when the ToDo list for a given thread is empty.

Returns: Boolean

Usage: Script or steady state.

Function Groups: Basic Module

Related to: Thread | ThreadHistory | ThreadList | ThreadName |

Format: ThreadIdle(ObjectValue)

Parameters:

ObjectValue

Required. The object value of the thread to monitor.

ThreadList

Description: Returns a two dimensional array containing the name and statement last executed by each VTScada thread.

Returns: Array

Usage: Script Only.

Function Groups: Basic Module, Software and Hardware

Related to: Profile | Thread | ThreadHistory | ThreadIdle | ThreadName |

Format: ThreadList()

Parameters: None

Comments: This function returns an allocated array such that the first row of the array (array[0][n]) contains the name of each thread, while the second row (array[1][n]) contains the statement most recently executed statement in that thread. The threads are ordered in the array in chronological order with the most recently started thread is at the beginning of the array.

Example:

```
If MatchKeys(1, "thread");
[
  allthreads = ThreadList();
]
```

ThreadName

Description: Returns the name of a thread.

Returns: Text

Usage: Script Only.

Function Groups: Basic Module, Software and Hardware

Related to: Profile | Thread | ThreadList | ThreadIdle | ThreadName |

Format: ThreadName(Instance)

Parameters:

Instance

Required. Any object value designating the thread for which the name is required.

Comments: This function returns the name of the thread in which Instance is executing.

Example:

```
If ! Valid(myThread);
[
  myThread = ThreadName(Self());
]
```

ThreadPriority

Description Allows advanced users to set a specified thread to one of six priorities, ranging from idle to time critical.

Warning: This statement is recommended for advanced users only.

Returns: Nothing

Usage: Script Only.

Function Groups: Basic Module

Related to:

Format: ThreadPriority(ObjectReference, Priority)

Parameters:

ObjectReference

Required. Indicates the thread to be modified.

Priority

Required. A value from -3 to 3 that indicates the priority of the thread. This may be one of:

Priority	Description
-3	THREAD_PRIORITY_IDLE
-2	THREAD_PRIORITY_LOWEST
-1	THREAD_PRIORITY_BELOW_NORMAL
0	THREAD_PRIORITY_NORMAL (Default)
1	THREAD_PRIORITY_ABOVE_NORMAL
2	THREAD_PRIORITY_HIGHEST
3	THREAD_PRIORITY_TIME_CRITICAL

Comments: All threads default to THREAD_PRIORITY_NORMAL. The operating system will place priority on threads operating at higher priorities over threads operating at lower priorities. The responsiveness of higher priority threads is therefore improved (at the expense of lower priority thread responsiveness).
Threads that share information amongst themselves may see performance drop if they don't all operate at the same priority; similarly high priority threads that perform extended operations may prevent lower priority threads from operating at all. Use of high priority threads is typically appropriate to short operations that need to be performed without delay or to processes that are intolerant of interruption.

Time

Description: Returns a formatted string for a time of day.

Returns: Text

Usage: Script or steady state.

Function Groups: Time and Date

Related to: Date | Now | Seconds

Format: Time(Sec, TimeForm [, Flags])

Parameters:

Sec

Required. Any numeric expression giving the number of seconds since mid night. The function, Now(1), is commonly used as it returns the current time of day, expressed in seconds.

TimeForm

Required. Any numeric expression giving the option for the time predefined Time Formats. If TimeForm is numeric, the format for the time will be interpreted. If TimeForm is a text value that does not resolve to a numeric, it is interpreted as a time formatting string as follows.

String	Description
"h"	Hours with no leading zero for single-digit hours; 12-hour clock.
"hh"	Hours with leading zero for single-digit hours; 12-hour clock.
"H"	Hours with no leading zero for single-digit hours; 24-hour clock.
HH	Hours with leading zero for single-digit hours; 24-hour clock.
"m"	Minutes with no leading zero for single-digit minutes.
"mm"	Minutes with leading zero for single-digit minutes.
"s"	Seconds with no leading zero for single-digit seconds.
"ss"	Seconds with leading zero for single-digit seconds.
"t"	One character time-marker string, such as A or P.
"tt"	Multi character time-marker string, such as AM or PM.

In the event that the TimeForm parameter does not resolve to either a numeric or text value, the system-configured time format, as specified

through the Windows Control Panel, is used. In this case, the Flags parameter is used to select from a number of options for the date.

Flags

An optional parameter that is only used in the event that the DateForm parameter does not resolve to a numeric or a text value. The Flags parameter may be set as follows to adjust the format of the date.

Value	Description
1	Do not generate minutes or seconds.
2	Generate minutes, but do not generate seconds.
8	Force 24-hour time format.

Note: The format string characters are case-sensitive. If you wish to include one of the formatting characters in your output string, then you must surround it with single quotation marks. For example, "h'h'" would display the current hour number in 12-hour clock format, with a lowercase h suffixed to it.

Comments: The Sec parameter may be negative, in which case it specifies the time before midnight. If greater than 86400, it specifies the time in the next day. The text string returned is at most 11 characters long.

Examples:

```
ZText(10, 10, Time(Now(1), 2), 0, 0);
time1 = Time(28800, 2);
time2 = Time(28800, 7);
time3 = Time(now(1), "m");
```

The first statement will display the current time in the upper right corner of the window in the form "08:24:13", and will update every second. The

value of time1, time2, and time3 will not be displayed, but will be set to "08:24:00", "08:24 AM" , and "24" respectively.

TimeArrived

Description: Indicates whether a given time has occurred.

Returns: Boolean

Usage: Steady State only. See: Rules for Usage.

Function Groups: Time and Date

Related to: CurrentTime

Format: TimeArrived(TriggerTime)

Parameters:

TriggerTime

Required. A UTC timestamp, specifying when the TimeArrived function will become true..

Example:

```
If TimeArrived(Timestamp);
[
   ...
]
```

This statement will trigger when the current time reaches the specified timestamp (Timestamp must be changed or invalidated in the corresponding script, otherwise the statement will trigger repeatedly). This example is exactly equivalent to the hypothetical statement:

```
If CurrentTime(1) >= Timestamp;
```

Except that CurrentTime may not be used in steady state.

TimeOut

Description: Returns true when the uninterrupted time that an expression is true reaches the specified value.

Returns: Boolean

Usage: Steady State only. See: Rules for Usage.

Function Groups: Time and Date

Related to: AbsTime | Now | RTimeOut

Format: TimeOut(Enable, Time)

Parameters:

Enable

Required. Any numeric expression giving the condition that results in the timer counting. When this parameter is true (not 0), the timer is "running." When this parameter is false (0), the timer stops and the timer is reset to 0.

Time

Required. Any numeric expression giving the time-out limit in seconds. When the cumulative time that Enable is true reaches this value, the function becomes true (1). If this value is 0 the function will trigger immediately and will continue to trigger as long as the state containing this function is active.

Comments: This function is reset when either parameter becomes invalid, the Enable becomes false, or when the state containing the function is started. When the function is reset, counting starts at 0 and the returned value is false (0). Note that this function is reset automatically when it occurs in a true action trigger or function parameter of a function which resets its parameters after evaluation (e.g. Latch, Toggle & Save).

Example:

```
ZText(10, 10 { Lower left corner of text },
      "Emergency!" { Text to display },
      Cond(Toggle(TimeOut(1, 0.3)), 12, 4
      { Toggle lt red/dark red every 0.3 secs }),
      0 { Default font });
```

This displays a message which flashes bright red and dark red every 0.3 seconds. Another common usage of the TimeOut function is as an action trigger for a script:

```
If TimeOut(1, 5) NextState;
[
   ...
]
```

This script will be executed 5 seconds after entering the state, and then a state change to NextState will occur.

TimeZone

Description: Returns information on the current time zone setting of the machine.

Returns: Varies

Usage: Script only.
May be used in optimized Tag Parameter Expressions.

Function Groups: Time and Date

Related to: Date | DateNum | Day | Month | Now | Seconds | CurrentTime | SetClock | Time | Today | Year | ConvertTimeStamp | TimeZoneList

Format: TimeZone(Option)

Parameters:

Option

Required. Any numeric expression giving the information to return as follows:

Option	Returns
0	Time displacement in seconds to UTC
1	Name of time zone (maximum of 32 characters) in "Daylight Time" or "Standard Time", which ever is applicable. Commonly used for display purposes.
2	Name of time zone (maximum of 32 characters) in "Standard Time" only. Commonly used as an input to ConvertTimestamp, but only if using an English OS.
3	A structure of time zone information: StdTimeZone (Standard time zone) ObservesDST (Boolean indicating Daylight Savings Time usage when true)

Comments:

Note that the first two options take account of whether daylight savings is in effect (assuming that that option has been selected on the machine). This means that not only the numeric time displacement, but also the name ("Daylight Time" versus "Standard Time") will vary according to the current date.

The output from the third option (2) is commonly used as an input to the ConvertTimestamp function since that function cannot use the adjusted timezone string from TimeZone(1). Only English is recognized.

Example:

```
If 1 Main;
[
```

```
  msg = Concat("It is now ", CurrentTime(), " ", TimeZone(1));
]
```

TimeZoneList

Description: Returns a list of time zones.

Returns: Array of text

Usage: Script Only.

Function Groups: Time and Date

Related to: ConvertTimeStamp | TimeZone

Format: TimeZoneList()

Parameters: None

Comments: Entries in the returned list may be used as parameters for the ConvertTimestamp function.
TimeZoneList produces a list of time zones in English, even on non-English versions of Windows.

Example:

```
Init [
  If 1 Main;
  [
    { Get a list of time zones }
    TZList = TimeZoneList();
  ]
]
```

Today

Description: Returns the current number of days since January 1, 1970.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Time and Date

Related to: Date | DateNum | Day | Month | Now | Seconds | Year

Format: Today()

Parameters: None

Example:

```
ZText(10, 10 { Lower left corner of text },
      Date(Today(), 4) { Display in Mmm dd, yyyy format },
      0 { Black color },
      0 { Default font });
```

This displays today's date in the upper left corner of the screen in the format "Mar 18, 1996".

TODBC

Description Performs an ODBC command; it is similar to ODBC except that it runs in its own thread (see "Comments" section for differences).

Returns Nothing (see parameters)

Usage Script Only.

Function Groups Database and Data Source, ODBC

Threaded Yes

Related to: ODBC | ODBCConfigureData | ODBCConnect | ODBCDisconnect | ODBCSources | ODBCStatus | ODBCTables | TODBCConnect | TODBCDisconnect

Format: TODBC(DB, SQLCommand, Attrib, Result, ErrorMsg, SQLState, ErrorCode [, QueryTimeout])

Parameters

DB

Required. An ODBC value for the ODBC database as returned by TODBCConnect or ODBCConnect.

SQLCommand

Required. Any text expression for the SQL command to perform on the ODBC database driver.

Attrib

Required. Any variable that will be set to an array that gives certain information regarding the table returned in the next parameter. The first dimension of the array contains the name of the column in the return table, while the second dimension contains the type - 0 for character data, 1 for numeric data.

Result

Required. Any variable that will be set to an array containing the data resulting from the SQL command. The format for the array is Result[Field][Record].

ErrorMsg

Required. Any variable that will contain the last error message returned by the function. If no errors occurred this parameter will be set to 0 to indicate the termination of the thread.

SQLState

Required. Any variable that will contain the SQL state that the statement was in when the last error occurred.

ErrorCode

Required. Any variable that will contain the native error code for the given driver and an error condition for the last error that occurred.

QueryTimeout

An optional parameter that sets the period (in seconds) which the driver will wait for a query request to complete. The default value of "0" indicates that there is no timeout. Please note that not all ODBC drivers support the optional QueryTimeout parameter; in particular, the Microsoft Access (.mdb) driver. If the driver does not support this option, then an error message will be returned by the statement. If this occurs, then the parameter should be left as Invalid to allow the statement to proceed.

Comments

This command will require a knowledge of SQL (Structured Query Language).
This function is typically used when a large amount of data is being processed. When TODBC is executed in its script, it starts its own thread and VTScada will continue executing. When the TODBC statement is finished executing, it will set the data in Attrib, give the resultant table in Result and set ErrorCode. Some SQL statements, such as the one for inserting a record into a table, do not return any data, in this case the only indication that the thread has terminated will be that ErrorCode will be set to a valid value - 0 if no errors occurred.
Notice that unlike ODBC, the resultant table is returned in one of the parameters, rather than as a return value and the last three parameters are not optional.
If any error, no matter how minor, occurs as a result of the SQL command having been executed, and the TODBCConnect or ODBCConnect that connected to the database had its Disconnect parameter set true, the value of DB will become invalid (i.e. the connection to the database will be dropped). This includes such trivial indiscretions as using an incorrect table name in the SQL command.

Differences between blocking and non-blocking ODBC calls:

- Prior to VTS version 10.0, executing one of the following blocking ODBC operations - ODBC, ODBCTables, ODBCConnect, ODBCDisconnect, ODBCBeginTrans, ODBCCommit or ODBCRollback would cause all VTScada script code execution and window painting to suspend until the operation was complete. From VTS 10.0 onwards, only the VTScada script thread making the call is suspended. All other threads and window painting continue to function.

- Prior to VTS version 10.0, a non-blocking ODBC operation was used to avoid this issue (same set of operations with a 'T' prepended). This leads to more complex script code insofar as you have to initiate an operation in a script and wait for completion in steady-state, but did allow script threads and window painting to continue. However, the Txxx operations were more time consuming to execute, as they each spin up a thread to execute the operation asynchronously. From VTS 10.0 onwards, these operations are more efficient - there is now no performance difference between the Txxx non-blocking variants and the blocking ones. As blocking ODBC calls no longer suspend all other threads, the only reason for using a Txxx variant call is where you wish to allow other script code statements in the same VTScada script thread to execute while the ODBC operation is processing.

Example:

```
If TimeOut(1, 3) { Execute SQL command every 3 seconds };
[
  TODBC(ODBCHandle { Handle to database },
       "SELECT ALL * FROM LogTasks" { SQL command },
       Attrib, ReturnArray { Results of command },
       ErrMsg, ErrState, ErrCode { Error details });
]
```

TODBCBeginTrans

Description: Indicates to an ODBC-compliant database that a transaction is to be started. TODBCBeginTrans is similar to ODBCBeginTrans, except that it runs in its own thread (see the Comments section for differences).

Returns: Nothing (see parameters)

Usage: Script Only.

Function Groups: Database and Data Source, ODBC

Threaded: Yes

Related to: ODBCBeginTrans | ODBCCommit | ODBCRollback | TODBCCommit | TODBCRollback

Format: TODBCBeginTrans(DB[, ErrorMsg, SQLState, ErrorCode])

Parameters:

DB

Required. A n ODBC value for the specified ODBC database as returned by ODBCConnect.

ErrorMsg

A parameter that will contain the last error message returned by the function.

SQLState

A parameter that will return the SQL state that the statement was in when the last error occurred.

ErrorCode

A variable that will contain the native error code for the given driver and an error condition for the last error that occurred.

Comments: TODBCBeginTrans indicates that a transaction is to be started on the specified ODBC database. The statement executes in its own thread, and completion is indicated by the ErrorCode parameter being set to a valid value, or to "0" in the case of no errors.
If any error, no matter how minor, occurs as a result of the statement, and the TODBCConnect or ODBCConnect that connected to the database had its Disconnect parameter set to true, the value of DB will become invalid (i.e. the connection to the database will be dropped).

Example:

```
StartBegin [
  If Condition WaitBegin;
  [
    TODBCBeginTrans(DB, Invalid, Invalid, ErrCode);
  ]
]
WaitBegin [
  If ErrCode WaitWork;
  [
    { Check Error code }
    …
    ErrCode = Invalid;
    { Do some transaction work }
    …
  ]
]
WaitWork [
  If ErrCode WaitCommit;
  [
    ErrCode = Invalid;
    TODBCCommit(DB, Invalid, Invalid, ErrCode);
  ]
]
WaitCommit [
  If ErrCode Done;
]
```

TODBCCommit

Description: Indicates to an ODBC-compliant database that a transaction is to be committed. TODBCCommit is similar to ODBCCommit, except that it runs in its own thread (see the Comments section for differences).

Returns: Nothing

Usage: Script Only.

Function Groups: Database and Data Source, ODBC

Threaded: Yes

Related to: ODBCBeginTrans | ODBCCommit | ODBCRollback | TODBCBeginTrans | TODBCRollback

Format: TODBCCommit(DB [, ErrorMsg, SQLState, ErrorCode])

Parameters:

DB

Required. An ODBC value for the specified ODBC data-

base as returned by ODBCConnect.

ErrorMsg

A parameter that will contain the last error message returned by the function.

SQLState

A parameter that will return the SQL state that the statement was in when the last error occurred.

ErrorCode

A parameter that is a variable that will contain the native error code for the given driver and an error condition for the last error that occurred.

Comments:

Commits a transaction defined as all the SQL statements since the transaction began. The statement executes in its own thread, and completion is indicated by the ErrorCode parameter being set to a valid value, or "0" in the case of no errors.

If any error, no matter how minor, occurs as a result of the statement, and the TODBCConnect or ODBCConnect that connected to the database had its Disconnect parameter set to true, the value of DB will become invalid (i.e. the connection to the database will be dropped).

Example:

```
StartBegin [
  If Condition WaitBegin;
  [
    TODBCBeginTrans(DB, Invalid, Invalid, ErrCode);
  ]
]
WaitBegin [
  If ErrCode WaitWork;
  [
    { Check Error code }
    …
    ErrCode = Invalid;
    { Do some transaction work }
    …
  ]
]
WaitWork [
```

```
  If ErrCode WaitCommit;
  [
    ErrCode = Invalid;
    TODBCCommit(DB, Invalid, Invalid, ErrCode);
  ]
]
WaitCommit [
  If ErrCode Done;
]
```

TODBCConnect

Description: Forms a connection to an ODBC database; it is similar to ODBCConnect except that it runs in its own thread (see "Comments" section for differences)

Returns: Nothing

Usage: Script Only.

Function Groups: Database and Data Source, ODBC

Threaded: Yes

Related to: ODBC | ODBCConfigureData | ODBCConnect | ODBCDisconnect | ODBCSources | ODBCStatus | ODBCTables | TODBC | TODBCDisconnect

Format: TODBCConnect(DSName, UserName, Password, DB, ErrorMsg, SQLState, ErrorCode [, Disconnect, LoginTimeout, ConnectionTimeout])

Parameters:

DSName

Required. Any text expression for the ODBC data source name, as configured in the ODBC setup menu under Microsoft Windows™.

UserName

Required. Any text expression for the ODBC login user name.

Password

Required. Any text expression for the ODBC login password.

DB

Required. An ODBC value for the ODBC database once a connection has been established.

ErrorMsg

Required. Any variable that will contain the last error message returned by the function. If no errors occurred this parameter will be set to 0 to indicate the termination of the thread.

SQLState

Required. Any variable that will contain the SQL state that the statement was in when the last error occurred.

ErrorCode

Required. Any variable that will contain the native error code for the given driver and an error condition for the last error that occurred.

Disconnect

An optional parameter which is any logical expression that determines how errors are to be handled. If true (non-0), the connection to the database will be disconnected should any error (no matter how minor) occur; if false (0) an error will not cause a disconnect to occur. The default value is false.

LoginTimeout

Sets the period (in seconds) which the driver will wait for a login request to complete. The default value of "0" indicates that there is no timeout.

ConnectionTimeout

Sets the period (in seconds) which the driver will wait for any request other than a query or login on the connection to complete. The default value of "0" indicates that there is no timeout.

Comments: When TODBCConnect is executed in its script, it starts its own thread and VTScada will continue executing. This is similar to ODBCConnect except that execution of the application is not suspended while waiting for the connection to the database (i.e. waiting for the return value to be set). Instead, execution continues and sometime in the future when the connection to the database is established, DB will be set to the ODBC value for the database. If any errors occurred, the error parameters, ErrorMsg, SQLState and ErrorCode, would be set at that time to indicate the nature of the error. ErrorCode will be set to zero if no error occurs.

64-bit VTScada. 64-bit VTScada is able to connect to either 64-bit or 32-bit ODBC data sources. ODBCConnect will first try to connect to the database through a 64-bit ODBC driver. If this fails for any reason it will then try the connection through a 32-bit ODBC driver. This means that any ODBC code that worked under 32-bit VTScada should not need to be modified for use with 64-bit VTScada, but 64-bit VTScada has the extra ability of being able to use 64-bit ODBC drivers.

Example:

```
Init[
  If 1 Wait;
  [
    TODBCConnect("VTS_data" { Driver name },
                 " "," " { No user name or password },
                   ODBCHandle { Handle to use },
                   ErrMsg, ErrState, ErrCode { Error details });
  ]
]
Wait[
  If TimeOut(1, Retry) Init { Retry ODBC every Retry seconds };
  If Valid(ODBCHandle) Main { If valid connection go to Main };
]
```

TODBCDisconnect

Description: Stops a connection to the ODBC database; it is similar to

ODBCDisconnect except that it runs in its own thread (see "Comments" section for differences).

Returns: Nothing

Usage: Script Only.

Function Groups: Database and Data Source, ODBC

Threaded: Yes

Related to: ODBC | ODBCConfigureData | ODBCConnect | ODBCDisconnect | ODBCSources | ODBCStatus | ODBCTables | TODBC | TODBCConnect

Format: TODBCDisconnect(DB)

Parameter:

DB

Required. An ODBC value for the ODBC database as returned by TODBCConnect or ODBCConnect.

Comments: When TODBCDisconnect is executed in its script, it starts its own thread which will not stop execution of VTScada, and which will exist until the connection to the database has been broken.

Example:

```
Main [
  If 1 Idle;
  [
    TODBC(ODBCHandle { Handle to database },
        "SELECT ALL * FROM Table1" { SQL command },
         Attrib, Result { Results of command },
         0, 0, 0 { No err details req'd });
  ]
]
Idle [
  If Valid(Result) Done;
  [
    TODBCDisconnect(ODBCHandle);
  ]
]
```

TODBCRollback

Description: Indicates to an ODBC-compliant database that a transaction is to be discarded. TODBCRollback is similar to ODBCRollback, except that it runs in its own thread (see the Comments section for differences).

Returns: Nothing

Usage: Script Only.

Function Groups: Database and Data Source, ODBC

Threaded: Yes

Related to: ODBCBeginTrans | ODBCCommit | ODBCRollback | TODBCBeginTrans | TODBCCommit

Format: TODBCRollback(DB[, ErrorMsg, SQLState, ErrorCode])

Parameters:

DB

Required. An ODBC value for the specified ODBC database as returned by ODBCConnect.

ErrorMsg

A parameter that will contain the last error message returned by the function.

SQLState

A parameter that will return the SQL state that the statement was in when the last error occurred.

ErrorCode

A variable that will contain the native error code for the given driver and an error condition for the last error that occurred.

Comments: Discards a transaction defined as all the SQL statements executed on a database since the transaction began.
If any error, no matter how minor, occurs as a result of the statement, and the TODBCConnect or ODBCConnect that connected to the database had its Disconnect parameter

set to true, the value of DB will become invalid (i.e. the connection to the database will be dropped).

Example:

```
StartBegin [
  If Condition WaitBegin;
  [
    TODBCBeginTrans(DB, Invalid, Invalid, ErrCode);
  ]
]
WaitBegin [
  If ErrCode WaitWork;
  [
    { Check Error code }
    …
    ErrCode = Invalid;
    { Do some transaction work }
    …
  ]
]
WaitWork [
  If ErrCode WaitEnd;
  [
    { Check error code, rollback if bad }
    IfElse(ErrCode != 0, Execute(
   ErrCode = Invalid;
     TODBCRollback(DB, Invalid, Invalid, ErrCode);
    );
    { Else } Execute(
   ErrCode = Invalid;
     TODBCCommit(DB, Invalid, Invalid, ErrCode);
    ));
  ]
]
WaitEnd [
If ErrCode Done;
]
```

Toggle

Description: Returns its previous status value except when its parameter changes from a false to a true, in which case it changes its value.

Returns: Boolean

Usage: Steady State only.

Function Groups: Variable

Related to: Latch

Format: Toggle(X)

Parameters:

X

Required. Any numeric expression giving the status value to use to cause the function value to change state.

Comments: This function starts in a state with its return value being the same as its parameter. If the parameter X changes from an in valid value to a valid false, the return value will be 0. If the parameter X changes from an invalid value to a valid true, the return value will be 1.
This function resets its parameters after they evaluate to true. This is significant only for functions which can be reset such as MatchKeys, TimeOut, Intgr and RTimeOut.

Example:

```
on = Toggle(MatchKeys(2, "A"));
```

The variable on begins set to 0. Pressing the "A" key will change it to 1; pressing "A" again will change it back to 0. Pressing "A" a third time will change it to 1 again, and so on.

Related Information:

See: "Latching and Resetting Functions" in the VTScada Programmer's Guide

ToLower

Description: Returns a text string with all the characters converted to lower case.

Returns: Text

Usage: Script or steady state.

Function Groups: String and Buffer

Related to: ToUpper

Format: ToLower(String)

Parameters:

String

Required. Any text expression giving the string to convert to all lower case characters.

Comments: The return value will contain all the same characters as the original string, except each character which is an upper case letter will be replaced with the corresponding lower case letter. All other characters remain unchanged.

Example:

```
lowerString = ToLower("Hello, Chum");
```

Upon execution of this statement, lowerString will contain the string "hello, chum".

ToolBar

(System Library)

Description: Draws and maintains a toolbar and its buttons.

Returns: Object Value

Usage: Steady State only.

Function Groups: Graphics

Related to: Bevel | CheckBox | Droplist | GridList | HScrollbar | Listbox | RadioButtons | Spinbox | SplitList | VScrollbar

Format: \System\Toolbar(Left, Top, Right, Data [, ParOffset, ParHasBevel])

Parameters:

Left

Required. The coordinate of the left edge of the toolbar.

Top

Required. The coordinate of the top of the toolbar.

Right

Required. The coordinate of the right edge of the toolbar.

Data

Required. A 2-dimensional array of button information.

ParOffset

An optional parameter that is the data index at which to start. The default for ParOffset is 0.

ParHasBevel

An optional parameter that can be set to FALSE (0) to draw a bevel around the toolbar, or TRUE (1) to inhibit bevel drawing. The default for ParHasBevel is 1.

Comments: Toolbar returns its object value when ready.

ToUpper

Description: Returns a text string with all the characters converted to upper case.

Returns: Text

Usage: Script or steady state.

Function Groups: String and Buffer

Related to: ToLower

Format: ToUpper(String)

Parameters:

String

Required. Any text expression giving the string to convert to all upper case characters.

Comments: The return value will contain all the same characters as the original string, except each character which is a lower case letter will be replaced with the corresponding upper case letter. All other characters remain unchanged.

Example:

```
upperString = ToUpper("Bye-bye Birdie");
```

Upon execution of this statement, upperString will contain the string "BYE-BYE BIRDIE".

Trajectory

Description: Move a Layered Graphic and return a Trajectory value.

Returns: Trajectory

Usage: Steady State only.

Function Groups: Graphics

Related to: Normalize | Path | Point

Format: Trajectory(Normalize, Path)

Parameters:

Normalize

Required. Any expression that returns a Normalize value. This gives the low and high scales for the animation.

Path

Required. Any expression which returns a Path value, which specifies the path along which the object moves.

Comments: This function must be called from within a window; that is to say, if for example the user uses the application template and has Graphics and Calculations as the two mod-

ules of the application, the Trajectory statement must go into Graphics.

Example:

```
bucketTraj = Trajectory(Normalize(bucketPos, 0, 100),
                        bucketPath);
```

If this Trajectory value is used in a Point, or in a layered graphics function, that graphic will move along the path bucketPath. If bucketPos is 0, the graphic will be displayed at the beginning of bucketPath; if bucketPos is 100, the graphic will be displayed at the end of bucketPath.

Transaction

(ODBC Manager Library)

Description: Launches a transaction in the specified database connection. The transaction takes care of its own shut-down process.

Returns: Nothing

Usage: Script Only.

Function Groups: ODBC

Related to: TransactionCached

Format: \ODBCManager\Transaction(TransObjPtr, ReadyPtr, TErrorPtr, CallerObj, UseTrans, DSN, UserName, Password)

Parameters:

TransObjPtr

Required. A Pointer to the transaction object

TReadyPtr

Required. A Pointer to a variable, used to set the ready status

TErrorPtr

Required. A Pointer to a variable, used to set the error

status

CallerObj

Required. The object value of the original calling module

UseTrans

Required. Set to true (1) to use BEGIN and END of transaction

DSN

Required. DSN of the database to start transaction within

UserName

Required. User name, if required by the database

Password

Required. Password, if required by the database

Comments: This module is a member of the ODBCManager Library, and must therefore be prefaced by \ODBCManager\, as shown in "Format" above.

TransactionCached

(ODBC Manager Library)

Description: Launches a transaction in the specified database connection. The transaction will be cached locally if it fails and then sent to the database after the next successful transaction. This module is designed to provide logging of values that must not be lost.

Returns: Returns an object value for the transaction.

Usage: Script Only.

Function Groups: ODBC

Related to: Transaction

Format: \ODBCManager\TransactionCached(ErrorPtr, CmdStr, DSN, UserName, Password[, BatchSize])

Parameters:

ErrorPtr

Required. A Pointer to the error status. Always set valid on completion.

CmdStr

Required. The SQL command to be processed within the transaction.

DSN

Required. DSN of the database within which to start the transaction.

UserName

Required. A user name, if required to connect to the database

Password

Required. A password, if required to connect to the database

BatchSize

Obsolete.

Comments: This module is a member of the ODBCManager Library, and must therefore be prefaced by \ODBCManager\, as shown in "Format" above.

TransferFields

Deprecated. Do not use in new code. (Alarm Manager module)

Description The TransferFields subroutine will transfer the values for each field into the returned FieldValues array. The values are found in the scope passed in using the variable names found in the FieldNames array.

Returns Numeric

Usage Script Only.

Function Groups Alarm

Related to:

Format: \AlarmManager\TransferFields(AlarmObject);

Parameters

AlarmObject

Required. The scope to use when searching for the variables to transfer.

Comments TransferFields always returns "0".

Trip

Note: Deprecated. Do not use in new code.

(Alarm Manager module)

Description: Tell the Alarm Manager to when a trip alarm event occurs. This subroutine will signal an alarm as unacknowledged.

Returns: Numeric

Usage: Script Only.

Function Groups: Alarm

Related to: Register (Alarm Manager) | CurrentTime

Format: \AlarmManager\Trip(AlarmObject[, EventTime]);

Parameters:

AlarmObject

Required. The object value for the alarm that was passed to the Register subroutine.

EventTime

Optional. The time stamp to use when adding this event to the alarm lists. If invalid or not defined, the default is CurrentTime().

Comments: The Trip subroutine always returns "0".

TRUE

Description: For use in expressions that perform Boolean logic. Using "TRUE" will make your code easier to read than using "1".

Returns: With no parameters, returns the value, 1. If given a parameter, this function will return a 1 or 0 depending on whether the parameter evaluates to TRUE or FALSE. Always returns 0 if the parameter is Invalid.

Usage: Script or steady state.

Function Groups: Logic Control

Related to: FALSE

Format: TRUE[(TestExpr)]

Parameters:

TestExpr

Optional. Any expression that evaluates to a 1 or 0 value. If no parameter is provided, then there is no need to include the parentheses.

Comments: This function exists to make your code more readable. It is equivalent to

```
PickValid(Cast(Parameter, 0) == 1, 0);
```

TServerList

Description: Executes in its own thread and creates a pointer to an array of all servers visible from this workstation; it returns a flag indicating its status upon completion.

Returns: Boolean

Usage: Script Only.

Function Groups: Database and Data Source, Network

Related to: ServerList | WKStaInfo

Format: TServerList(Result, Domain)

Parameters:

Result

Required. Any variable in which the resultant one-dimensional array of servers or an error code will be returned.

Domain

Required. Any text expression for the domain. If this is invalid, the current domain will be used.

Comments: When TServerList is executed in its script, it creates its own thread and VTScada continues executing. When it is finished executing, it will store resultant data in Result (unlike ServerList which returns its result). Result will be set to an error code if any network problems were encountered, or if Domain was not found on the network.
This function returns 1 if the thread was successfully started and 0 otherwise.

Example:

```
If ! Valid(serverArray) Wait;
[
  started = TServerList(serverArray, WkStaInfo(2));
]
```

U Functions

The sections that follow identify all VTScada functions beginning with "U".

UIErrorToText

Security Manager Module

Description Returns a text string corresponding to the error code provided.

Returns String

Usage Script or steady state.

Related to: GetAccountID | GetAccountInfo | GetFullName | GetGroupName | GetUserName | IsLoggedOn | IsSecured | IsSuspended | SecurityCheck |

Format: \SecurityManager\UIErrorToText(ErrCode)

Parameters

ErrCode

One of the #SMAPIErrxxx error codes.

Comments None.

Related Information:

See "Security Manager Return Codes" in the VTScada Programmer's Guide.

Unpack

Description: Unpacks a set of values from a stream into a single dimensional array or a set of variables referenced by object parameters, and returns the number of items unpacked.

Returns: Numeric

Usage: Script Only.

Function Groups: Array, Stream and Socket

Related to: Pack

Format: Unpack(Data, Start, End, Stream[, TruncateAt, DataLength, MirrorKey])

Parameters:

Data

Required. An object value or an array containing the data or the object value of the module whose parameters address variables into which to store the data.

For example, if you have 5 numeric values to unpack, you would allocate a 1-dimensional array, 5 elements in length. You would pass this array to Unpack's Data parameter, and specify that you wish to unpack from subscript 1 to 5. Refer to the example section for more information.

Start

Required. The starting array index (zero-based), or parameter number (one-based) of the data to unpack.

End

Required. The last array index, or parameter number of the data to unpack.

Stream

Required. A variable holding the stream that contains the data to be unpacked. The stream must have been generated with the Pack function.

TruncateAt

An optional parameter. If present specifies that all text values and stream values unpacked from the stream will be truncated after the number of bytes specified in this parameter.

DataLength

An optional parameter. If present, specifies an array into which will be stored the length of each text and stream value unpacked from the source stream.
Each length is stored at the array index corresponding to the index of the value itself. For example, if the value that would go in the destination array at index 5 were a text value, its length would be stored at index 5 in the array addressed by this parameter.
Unpacked values of numeric type have Invalid stored in their entry in this array.

MirrorKey

Optional. A short name or number. Must be used if the data was packed using a Key.
There is no requirement for the Pack Key parameter and the Unpack MirrorKey parameter to have the same number of elements. All that is required is that the Unpack MirrorKey dictionary has all of the structures that are in that particular packed stream. If the Key doesn't have the structure that was packed, the returned data is a simple array rather than a structure.

Comments: This function returns the number of values unpacked.
If the Data parameter is an array, the data from the stream will be unpacked into that array.
If the Data parameter is an object value, the parameters of that object must contain pointers to variables into which the data from the stream will be unpacked.
If the Pack function used the optional parameter, Key, to pack a record into a smaller stream, then Unpack must be able to provide a mirrored version of that dictionary in order to access the data in the structure.

Example:

If you wish to pack 2 3-dimensional arrays, you would allocate a 1-dimensional array, 2 elements in length. You would then assign the 3-dimensional arrays to the 1-dimensional array elements, and then Pack the 1-dimensional array.
For example:

```
Array1D = New(2);
Array1D[0] = Array3D_1;
Array1D[1] = Array3D_2;
Pack(Array1D, 0, 1, &Stream);
```

Unpacking this is almost an identical operation:

```
Array1D = New(2);
UnPack(Array1D, 0, 1, Stream);
Array3D_1 = Array1D[0];
Array3D_2 = Array1D[1];
```

All you need to know is how many things you wish to pack and unpack, not the sizes or types of the things (a ValueType(Array1D[0]) will tell you the type). As the stream position is moved after each pack or unpack, you can simply unpack one item at a time and check the return value of UnPack.

Example 2:

In this example, the presence of a Key parameter allows the example to be packed into a stream that is less than half the size that would be obtained without the Key parameter. (Exact number depending on the names used in the structure definitions.)
"Record", "Values", and "AA_Info" are all STRUCTs.

```
Rec = Record( GetGUID(1)             { GUID       },
              1                      { Priority   },
              Values(87, 54, "feet") { Values     },
              System\MakeDictionary("AA",
                            AA_Info("AAData1", 2)) { Extensions });

{ Omitted: Place record in ArrayToPack }

Key = \System\MakeDictionary("Record", 0,
                             "Values", 1,
                             "AA_Info", "AA")

Pack(ArrayToPack, 0, 0, &PackStream, Key);

{ Omitted: Rewind stream }

MirrorKey = \System\MakeDictionary("0", Record,
                                   "1", Values,
                                   "AA", AA_Info));

UnPack(UnpackedArray, 0, 0, PackStream, Invalid, Invalid, MirrorKey);
```

UnpackData

(RPC Manager Library)

Description: This method unpacks a stream into an array or set of mod-

ule instance parameters. Subroutine call only.

Returns: Nothing (see parameters)

Usage: Script Only.

Function Groups: Array, Stream and Socket

Related to:

Format: \RPCManager\UnpackData(Source, Start, End, Stream [, TruncLen, DataLen]));

Parameters:

Source

Required. The array to contain the data or the object value of the module whose parameters address variables into which to store the data.

Start

Required. The starting array index or parameter number of the data to unpack (zero-based).

End

Required. The last array index or parameter number of the data to unpack (zero-based).

Stream

Required. The stream that contains the data to be unpacked. The stream must have been generated with the PackData function.

TruncLen

An optional parameter that specifies that all text values and stream values unpacked from the stream will be truncated after the number of bytes specified in this parameter.

DataLen

An optional parameter that specifies an array into which the length of each text and stream value

unpacked from the source stream will be stored. Each length is stored at the array index corresponding to the index of the value itself.
For example, if the value which would go in the destination array at index 5 were a text value, its length would be stored at index 5 in the array addressed by this parameter.
Unpacked values of numeric type have Invalid stored in their entry in this array.

Comments: This subroutine is a member of the RPC Manager's Library, and must therefore be prefaced by \RPCManager\, as shown in the "Format" section. If the application you are developing is a script application, the subroutine call must be prefaced by System\RPCManager\, and the System variable must be declared in AppRoot.SRC.
The stream contents to be unpacked by the UnpackData method must have been packed by the PackData method.
If the Source parameter is an array, the data from the stream will be unpacked into that array. If the Source parameter is an object value, the parameters of that object must contain pointers to variables into which the data from the stream will be unpacked.
From VTS 5.20 onwards, this method is a wrapper for the Unpack() statement. Prior to that release, the Unpack() statement was effectively coded in script by this method. New code should use the Unpack() statement, rather than this method.

Related Functions:

Pack | Unpack

UnpackParms

(RPC Manager Library)

Description: This method unpacks a stream into the supplied parameters. Subroutine call only.

Returns: Nothing

Usage: Script Only.

Function Groups: Network, Stream and Socket

Related to: PackParms

Format: \RPCManager\UnpackParms(Stream [, &P1, &P2, ...]);

Parameters:

Stream

Required. A stream produced by PackParms.

P1, P2, ...

Optional parameters that are pointers to variable instances into which the values that were packed into the stream will be unpacked. Up to 100 parameters are allowed.

Comments: This subroutine is a member of the RPC Manager's Library, and must therefore be prefaced by \RPCManager\, as shown in the "Format" section. If the application you are developing is a script application, the subroutine call must be prefaced by System\RPCManager\, and the System variable must be declared in AppRoot.SRC.
The stream contents must have been packed by the PackParms method.

Unregister (Alarm Manager)

Note: Deprecated. Do not use in new code.

(Alarm Manager module)

Description: Notify the Alarm Manager that an alarm has been removed. This will not generate an alarm; it only removes it from the list of all configured alarms.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Alarm

Related to:

Format: \AlarmManager\Unregister(AlarmObject);

Parameters:

AlarmObject

Required. The object value of the alarm to remove from the Alarm Manager database.

Comments: The Unregister subroutine always returns "0".

UnselectGraphics

Description: Will deselect all of the graphics in the specified window.

Returns: Nothing

Usage: Script or steady state.

Function Groups: Graphics, Window

Related to: SelectArea | SelectDAG | SelectGraphic | UnselectObject

Format: UnselectGraphics(Window)

Parameters:

Window

Required. Any expression which gives the object value of any module instance which is drawn in the window.

Example:

```
If ZButton(30, 100, 130, 70, "Deselect", 1);
[
  UnselectGraphics(Self());
]
```

If the button that is displayed in the upper left corner of the window is pressed, all graphic objects I that window that were selected will become deselected.

UnselectObject

Description: Will deselect a statement in the context supplied.

Returns: Nothing

Usage: Script or steady state.

Function Groups: Graphics

Related to: SelectArea | SelectDAG | SelectGraphic | UnselectGraphics

Format: UnselectObject(Object, Statement)

Parameters:

Object

Required. Any expression which gives the object value for the instance where the graphic is to be deselected.

Statement

Required. The value of the statement which is to be deselected.

UnTransform

Description: Will undo a previous transform so that the module instance and everything it has called will not be transformed.

Returns: Nothing

Usage: Script Only.

Function Groups: Graphics, Advanced Module

Related to: GUITransform | GetXformRefBox

Format: Untransform(Object)

Parameters:

Object

Required. Any expression which gives the object value for the instance to be untransformed.

Comments: This function is used to allow a module instance to draw in the default window coordinates even if the instance is contained within a transform.
This statement may only appear in a script.

Example:
Suppose that the System module calls module Pump inside of a transform as follows:

```
GUITransform(0, 90, 90, 0 { Bounding box for transform },
             1, 1, 1, 1, 1 { No scaling },
             0, 0 { No trajectory or rotation },
             1, 0 { Graphics visible; reserved },
             0, 0, 0 { Cannot be focused },
             Pump() { Module to transform });
```

Now let us further suppose that Pump calls a module named Reservoir. This module will also be affected by the transform of its caller unless it uses Untransform to nullify the effects of the transform performed on Pump. The following is the first state of module Reservoir:

```
Init [
  If 1 Main;
  [
    Untransform(Self());
  ]
]
```

With this as its first state, Reservoir will now be drawn in the window as if it were called directly from System, that is to say, it will not be affected whatsoever by the transform that resizes its caller, Pump. Notice that the Untransform function need only be called once; no matter what further state changes occur within Reservoir, it will remain unaffected by the transform that affects Pump.

UpdateCoordinates

Description: Will update a graphic statement's coordinates to the document file in which it is specified.

Returns: Nothing

Usage: Script Only.

Function Groups: Graphics, Advanced Module

Related to:

Format: UpdateCoordinates(Statement)

Parameters:

Statement

Required. Any expression which gives a statement pointer or code value for the graphic statement which is to be updated.

Comments: It should be noted that once this graphic statement is recompiled that any new instances of that statement will have the new coordinate information stored in them.

UserCredChange

Security Manager Module

Description: The return value will increment each time there is a change in the user session's logged-in user or their password.

Returns: Integer

Usage: Steady State only.

Related to: AlternateIdCheck | AlternateLogoff | AlternateLogon | Authenticate | LogOff | QuietLogon | UserLogonDialog

Format: \SecurityManager\UserCredChange(Session);

Parameters:

Session

The object value of the user session.

Comments: Initially returns zero
For VIC sessions, this additionally increments its return value at twice the rate specified by the SessionTokenTimeout configuration setting. This is used by internal system components to refresh a VIC's security session token.

UserLogonDialog

Security Manager Module

Description: Launches the Logon dialog

Returns: Nothing

Usage: Script Only.

Related to: AlternateIdCheck | AlternateLogoff | AlternateLogon | Authenticate | LogOff | QuietLogon | UserCredChange

Format: \SecurityManager\UserLogonDialog([Device, Namespace, Center, Embed, DialogX, DialogY, DialogDirection, pULDObj, AppLayer]);

Parameters:

Device

Name of device logging on

NameSpace

Default setting for the Group name (optional)

Center

TRUE to center dialog on screen (default=0)

Embed

TRUE to draw the dialog embedded in a page
FALSE to show the dialog in a window (default)

DialogX

X coordinate to base dialog position upon

DialogY

Y coordinate to base dialog position upon

DialogDirection

Numeric code giving the direction from (DialogX,DialogY) to displace:

Code value	Direction to displace
0	Down and Right
1	Down and Left
2	Up and Right
3	Up and Left

pULDObj

Returns a pointer that is set to the logon dialog

AppLayer

Layer value of the calling Layer. This may differ from LayerRoot when shared security is enabled.

Comments: None

V Functions

The sections that follow identify all VTScada functions beginning with "V".

Valid

Description Returns true if the parameter is valid.

Returns Boolean

Usage Script or steady state.

Function Groups Variable

Related to: Invalid

Format: Valid(X)

Parameters:

X

Required. Any expression to be tested for validity.

Comments This function always returns a valid value.

Examples:

```
a = Valid(1.23);
b = Valid("Help");
c = Valid(Invalid);
d = Valid(12.3 / 0);
```

The values for a, b, c and d will be 1, 1, 0 and 0 respectively.
A common use for this function is as an action trigger to prevent "if 1" conditions:

```
If ! Valid(startTime);
[
  startTime = Seconds();
]
```

This will cause the script to execute once and once only; startTime will be set to the time (in seconds since midnight) that the script was executed.

ValidateEmailAddrs

Description: This subroutine validates a string of email addresses, and returns TRUE if the email addresses in the string are syntactically valid, or FALSE if they are not.

Returns: Boolean

Usage: Script or steady state.

Function Groups: Email

Related to: SendMail

Format: \ValidateEmailAddrs(EmailAddress)

Parameters:

EmailAddress

Required. The string of email addresses to check.

Comments: Multiple addresses are assumed to be separated by semi-

colons in the input string. This subroutine returns TRUE if the email addresses in the string are syntactically valid, otherwise, FALSE is returned.

ValidateEmailAddrs checks that each address has an @ symbol that is not the first or last character, and that each address has a dot appearing at least 2 characters after the @ symbol, with no dot appearing immediately after the @ sign, and no dot as last character.

ValueType

Description: Returns the type of value passed to it.

Warning: This function should be used by advanced users only.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Variable

Related to: Cast

Format: ValueType(Val)

Parameters:

Val

Required. Any expression.

Comments: This returns the type of the value passed in the parameter

Example:

```
type = ValueType(2.31);
```

The value of type will be 3.

Related Information:

VTScada Value Types - Numeric Reference

VarAttributes

Description: Returns the attributes bit field of a variable.

Warning: This function should be used by advanced users only.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Compilation and On-Line Modifications, Variable

Related to:

Format: VarAttributes(Variable)

Parameters:

Variable

Required. Any variable value expression.

Comments: The return value indicates the attributes of the variable by setting certain bits in the value as follows:

Return Value	Bit No.	Attribute
1	0	Array
2	1	Shared
4	2	Persistent
8	3	Module
16	4	Parameter
32	5	Constant
64	6	<obsolete>
128	7	<obsolete>
256	8	Temporary
512	9	Protected

Variable

Description: Accesses a variable by its text name; its return value is optional.

Returns: Varies – see comments

Usage: Script Only.

Function Groups: Variable

Related to: FindVariable | Scope | SetDefault

Format: Variable(Name)

Parameters:

Name

Required. Any text expression giving the name of the variable.

Comments: This function can be used on the left side of an assignment, in which case the value will be assigned to the variable named in the Name parameter. This function can be used for debugging, or to create sophisticated data logging and monitoring packages which access any variable by its typed-in name.
Note that the string containing the name may not contain leading or trailing spaces, or square brackets [ ].

This function is the same as the '\' operator, when the '\' operator is before a single operand. (\Member).

This function is able to resolve variables that have the PROTECTED attribute.

LocalVariable
A related function, LocalVariable() exists, but will be used rarely. No documentation is provided beyond

the following note:
LocalVariable(Name) is the same as Scope(Self, Name, TRUE). This expression is useful only in the case that there is a value containing the name of a variable, which is added after the module is compiled and is referenced within the local module. LocalVariable is the same as the '.' operator. .Member (i.e. without an object before the dot) compiles to LocalVariable("Member"), which is the equivalent of Variable("Member", TRUE).

Example:

```
trendValue = Variable(trendPointName);
```

The variable trendValue is set equal to the value of the variable named by variable trendPointName. The variable trendPointName is a text variable, which might hold the text name of a point entered by the operator from the keyboard.

```
Variable(motorSPName) = newSP;
```

This shows how to set the value of a variable given its text name. The variable motorSPName is a text variable that holds the name of the destination variable.

VariableClass

Description: Returns the class of a variable.

Returns: Numeric

Usage: Script Only.

Function Groups: Variable

Related to: SetVariableClass

Format: VariableClass(Variable)

Parameters:

Variable

Required. Any expression for the variable value.

Comments: This function may only appear in a script.

Example:

The following sets the variable "class" to the class of variable newVar.

```
If ! Valid(class);
[
  class = VariableClass(FindVariable("newVar", Self(), 0, 1));
]
```

Find the class of Alarm Priority tags:

```
Class = VariableClass(FindVariable("AlarmPriority", \Code, 0, 0));
```

Variance

Description: Returns the statistical sample variance for a subsection of an array.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Array, Generic Math

Related to: AMax | AMin | AValid | FiltHigh | FiltLow | FitOffset | FitSlope | Mean | SDev | Sum

Format: Variance(ArrayElem, N)

Parameters:

ArrayElem

Required. Any numeric array element giving the starting point in the array for the computation. The subscript for the array may be any numeric expression. If processing a multidimensional array, the usual rules apply to decide which dimension should be examined.

N

Required. Any numeric expression giving the number

of array elements to compute. If N extends past the upper bound of the lowest array dimension, this computation will "wrap-around" and resume at element 0, until N elements have been processed.

Comments: The invalid parameters are not skipped over, but they are not included in the calculation. The function returns an invalid result if either of its parameters is invalid or if there are less than two valid numerical array elements in the specified range.

Example:

```
weight[0] = Invalid();
weight[1] = 0;
weight[2] = 1;
weight[3] = 3;
weight[4] = 1;
weight[5] = 5;
prodVariance = Variance(weight[4] { Starting array element },
4 { No. of elements to use });
```

The variable prodVariance will be the variance of elements 4, 5, 0 and 1 of the array weight, which is 7. Since element 0 is an invalid element, the variance will be calculated using only the 3 valid elements.

Version

Description: Returns the version number of the copy of VTScada currently running.

Returns: Text

Usage: Script only.
May be used in optimized Tag Parameter Expressions.

Function Groups: Software and Hardware

Related to: SerialNum | VersionRequired

Format: Version([MiniDumpHandle, InfoType])

Parameters:

MiniDumpHandle

For advanced use only. If provided, will return information about the version number or bit-width of a mini dump file. Data is stored in a MiniDumpHandle (value type: 42)
It is recommended that you simply use INVALID as a placeholder for this parameter when the second parameter is required.

InfoType

If 0 or omitted, Version simply returns the VTScada version number.
If 1, will return 32 for 32-bit VTScada or 64 for 64-bit VTScada, instead of the version number

Comments: This function can be used to perform various tasks based on the version number of VTScada.

Example:

```
ZText(10, 10, Concat("VTScada version ", Version()), 0, 0);
```

This displays the version number of VTScada.

VersionRequired

Description: Returns the version number of VTScada that is required to execute any .RUN files loaded since the last execution of this statement.

Returns: Text

Usage: Script only.
May be used in optimized Tag Parameter Expressions.

Function Groups: Software and Hardware

Related to: Version

Format: VersionRequired()

Parameters: None

Comments: This function returns the minimum version of VTScada

required to execute any and all modules (.RUN files) loaded since the last call to this function, or if no other call was made, since startup. Once the call is made to this function, it will reset the version to invalid until another module is loaded.

Example:

```
If 1 Next;
[
  minVersion = VersionRequired();
  IfThen(minVersion > Version(),
  msg = Concat(".RUN files require version ", minVersion);
  msgOpen = 1;
  );
]
```

This script tests the version number of VTScada required to execute .RUN files loaded prior to this point in the application and executes a series of statements if they are greater than the version of VTScada that is currently running.

Vertex

Description: Returns a Vertex value, which is a collection of 3 points and a mode.

Returns: Vertex

Usage: Steady State only.

Function Groups: Graphics

Related to: Path | Point

Format: Vertex(Mode, CenterPoint, InHandlePoint, OutHandlePoint)

Parameters:

Mode

Required. Any numeric constant that specifies the behavior of the handle points, as shown in the following table

Mode	Handle Point Behavior
0	Rectangular – handles are ignored
1	Cusp – no restrictions on handle points
2	Reserved for future use
3	Reserved for future use
4	Manhattan[1] – handles are ignored. Right angles between this vertex and neighboring vertices are preserved, enforcing horizontal or vertical lines.

CenterPoint

Required. Any expression that returns a point object. This is the center point and location of this Vertex.

InHandlePoint

Required. Any expression that returns a point.

OutHandlePoint

Required. Any expression that returns a point.

Comments:

Vertex values are used in paths to specify points along the path. For each vertex, the center point defines a point on the path. Each line segment or Bezier curve along the path is defined by two vertices, one at each end. The vertices' center point is the end point of the Bezier curve. The OutHandlePoint of one vertex and the InHandlePoint of the other vertex define the Bezier curve shape points. So, each Bezier curve is defined by 4 points: the two end points, and two 'handle' points.

---

[1]Meaning that all lines are horizontal or vertical. Inspired by a skyline of tall, rectangular buildings.

Vertex values are also used in the graphics functions GUIArc, GUIChord, and GUIPie to determine the start and end angles for those graphics functions. The angle from the center point to the InHandlePoint defines the starting angle. The angle from the center point to the OutHandlePoint defines the ending angle

Example:

```
GUIArc(10, 100, 100, 10 { Bounding box for the arc },
       1, 1, 1, 1, 1 { No scaling },
       0, 0 { No trajectory or rotation },
       1, 0 { Arc is visible; reserved },
       0, 0, 0 { Cannot be focused/selected },
       Pen(12, 1, 1) { Solid lt red line, 1 pixel wide },
       Vertex(0 { Rectangular mode },
       Point(50, 50, Invalid, Invalid) { Arc center },
       Point(50, 0, Invalid, Invalid) { Start angle },
       Point(0, 50, Invalid, Invalid) { End angle }));
```

The Vertex function in this case defines the center, starting angle and ending angle for the arc. The center point is (50, 50); the starting angle is defined by the center point and a point that lies directly above it, and the ending angle is defined by the center point and a point that lies directly to the left of it. The result is a small arc that starts at the 12 o'clock position and runs counter-clockwise to the 9 o'clock position.

VICInfo

Description: Provides information about the VTScada Internet Clients connected to the machine where this code is run.

Returns: Array – see comments

Usage: Script only.
May be used in optimized Tag Parameter Expressions.

Function Groups: VTScada Internet Client

Related to: VICMessage

Format: VICInfo()

Parameters: None

Comments: The return value of VICInfo is an array of information. Each row [leftmost subscript] contains the information for one session, and each column within the row contains the fields of session information, as follows:

[n][0] Object value of the root (BrowserClient) module instance for this session.

[n][1] VIC process session ID [a 16-byte binary GUID]. This is guaranteed to be unique per VIC process. i.e. two sessions from the same VIC process will have the same VIC instance ID.

[n][2] IP of the remote computer, from the server's point of view [note this may be the IP of a gateway or proxy that the remote computer connects via].

[n][3] Total bytes transmitted to the VIC in this session.

[n][4] Total bytes received from the VIC in this session.

[n][5] Approximate round trip time to the VIC, in milliseconds.

[n][6] Uncompressed bytes transmitted to the VIC.

[n][7] Session identifier for the connection to the server.

VICMessage

Description: Transmits a message to one or all currently connected VTScada Internet Client sessions. The message is displayed in a dialog box on the VIC computer.

Returns: Nothing

Usage: Script Only.

Function Groups: VTScada Internet Client

Related to: VICInfo

Format: VICMessage(Object, Message)

Parameters:

Object

Required. Object value of the root (BrowserClient) module instance for the VIC session to receive the message. If this parameter is not a valid object value, the message is broadcast to all VIC sessions.

Message

Required. The textual message to be sent to the VIC recipients.

Comments: There is no return value for this statement.
The Object value is typically retrieved from the return value of a VICInfo statement.

VoiceTalk

Description: Opens and returns a handle to a SAPI text-to-speech stream.

Returns: Stream handle

Usage: Steady State only.

Function Groups: Speech and Sound

Related to: Configure | GetDevices | GetVoices | Reset | ShowLexicon | Speak

Format: \VoiceTalk([SpeakCount, BookmarkNum])

Parameters:

SpeakCount

An optional parameter that is any variable in which the number of outstanding VoiceTalk\Speak requests (those that have not yet finished speaking) on this stream will be maintained.
When this parameter is set to zero, this SAPI text-to-

speech stream is not speaking (i.e. it is initially set to zero). If this parameter is omitted or Invalid, no speak count will be maintained.
Note that SAPI could still be speaking on another stream when this count is zero on one stream.

BookmarkNum

An optional parameter that is any variable that will be set to the value of the last text bookmark encountered by the speech engine on this stream (a bookmark is denoted by the XML tag <BOOKMARK MARK-K="bookmark"/>). It is initially set to Invalid.

Comments: This function returns the error code resulting from issuing the command to the speech engine, or zero if no error was encountered.
This function is not threaded; however, it creates a thread inside of which the handle referring to the text-to-speech stream is accessed. All other speech functions on this stream do not create their own thread, but will execute in the thread created by this function. This thread will exist for as long as the VoiceTalk statement remains active (i.e. until a state change occurs). For this reason, the state containing the VoiceTalk call must remain active until all other speech statements have finished executing. There can be multiple SAPI text-to-speech streams open at any time.

Example:

```
speechHandle = \VoiceTalk();
```

This will create a SAPI text-to-speech stream that will maintain neither speak count nor bookmark positions.

```
speechHandle = \VoiceTalk(sCount, bNum);
```

This will create a SAPI text-to-speech stream that will maintain both a speak count and bookmark positions.

```
speechHandle = \VoiceTalk(invalid, bNum);
```

This will create a SAPI text-to-speech stream that will maintain bookmark positions only.
The following list consists of the methods that a VoiceTalk stream provides for external use.

VScrollbar

(System Library)

Description: Draws a vertical scroll bar and returns its position.

Returns: Numeric

Usage: Steady State only.

Function Groups: Graphics

Related to: Bevel | CheckBox | ColorSelect | Droplist | Edit | HScrollbar | Listbox | RadioButtons | SplitList | Spinbox

Format: \System\VScrollbar(Left, Bottom, Height, Steps, PageLen[, Offset, StepSize, MouseWheelInputObj])

Parameters:

Left

Required. Any numeric expression for the left side coordinate of the scrollbar.

Bottom

Required. Any numeric expression for the bottom coordinate of the scrollbar.

Height

Required. Any numeric expression for the height of the scrollbar in pixels.

Steps

Required. Any numeric expression giving the number of steps in the scrollbar.

PageLen

Required. Any numeric expression giving the number

of steps in each page.

Offset

Optional. A variable whose value gives the position of the thumb tab. Defaults to 0 unless otherwise specified.

StepSize

Optional. Any numeric expression giving the number of lines to scroll through when the user clicks on an arrow. Defaults to 1 if not specified.

MouseWheelInputObj

The Window object that should capture mouse wheel messages for this VScrollbar. Every VScrollbar module requires a unique source window in order for mouse wheel scrolling to function properly. On the server and the VIC, this parameter is used so that scroll wheel events that happen over MouseWheelInputObj's window are redirected to the VScrollbar.
On the Anywhere Client, in addition to redirecting scroll wheel events as above (if the client's platform happens to have a mouse), this parameter is also necessary to redirect vertical touch panning events over a given region to the VScrollbar. This is of increased importance because on most touch screen devices, you cannot touch and drag the scrollbar directly, so this kind of touch redirection is often the only way to scroll a VScrollbar.

Comments:

This module is a member of the System Library, and must therefore be prefaced by \System\, as shown in the "Format" section. If you are developing a script application, use "System\..." rather than "\System\..." in the function

call.
Steps can be calculated by subtracting the number of visible items from the total number of scrollable items, while PageLen is equal to the number of visible items.
Custom script graphics using the VScrollbar module directly need to pass in a window object to serve as source for the mouse wheel messages. Typically this would be the window that the VScrollbar module is controlling. Note that the VScrollbar module itself uses a native windows scrollbar which will therefore work automatically without any script changes, so long as the mouse pointer is over the actual scrollbar.

Example:

```
\System\VScrollBar(VStatus(Self, 11) - VStatus(Self, 21)
                { Left },
                VStatus(Self, 12) - 1 { Bottom },
                VStatus(Self, 12) { Height },
                Length - LinesVisible { Total steps },
                LinesVisible { Steps in page },
                Offset { Thumb tab pos });
```

VStatus

Description: Returns the video board and screen characteristics for VTScada.

Returns: Varies – see table in Option parameter

Usage: Script or steady state.

Function Groups: Software and Hardware, Window

Related to: Coordinates | PalStatus

Format: VStatus(Window, Option)

Parameters:

Window

Required. The object value of the window to request information on. This may be invalid and is therefore

ignored for Options which do not depend upon a particular window.

Option

Option	Video Characteristic
0	Reserved for future use
1	Reserved for future use
2	Reserved for future use
3	X Coordinate of left side of the window. This is the client area of the window and does not include the title or borders.
4	Y Coordinate of bottom side of the window. This is the client area of the window and does not include the title or borders.
5	X Coordinate of right side of the window. This is the client area of the window and does not include the title or borders.
6	Y Coordinate of top side of the window. This is the client area of the window and does not include the title or borders.
7	Number of horizontal pixels of resolution of the virtual client area of the window. This doesn't include the window border. Shrinking the window to make scroll bars appear doesn't affect this number.
8	Number of vertical pixels of resolution of the virtual client area of the window. This doesn't include the title or border of the window. Shrinking the window to make scroll bars appear doesn't affect this number.

Example:

```
ZGrid(VStatus(Self(), 3) { Left edge of window },
      VStatus(Self(), 4) { Bottom edge of window },
      VStatus(Self(), 5) { Right edge of window },
      VStatus(Self(), 6) { Top edge of window },
      12 { Grid is red },
      10, 10 { X and Y spacing });
```

This draws a 10 by 10 red color dotted grid in the entire area of the current window.

W Functions

The sections that follow identify all VTScada functions beginning with "W".

Watch

Description: Watches its parameters and returns true when any of their types or values change.

Returns: Boolean

Usage: Steady State only. See: Rules for Usage.

Function Groups: Variable

Related to: Change | Edge | WatchArray

Format: Watch(Start[, Parm1 [, Parm2, ...])

Parameters:

Start

Required. Any expression which evaluates to a false (0) or true (non-0) value. This will be the initial return value of the function.

Parm1, Parm2, ...

Are any number of optional variables that are to be monitored by this function.

Comments: This is a reset-able function whose initial return value will be set by the first parameter. This parameter is ignored after the initial evaluation of the function.
The Watch function will not be triggered by a variable whose value has been set to its existing value. That is to say, if a variable has a value of 3 and it is set in the code again to a value of 3, then there is no change and this function will not be triggered.
Only the first parameter (Start) is mandatory. Use Watch(1) to trigger a script once only on the first entry to a state.

Examples:

```
If Watch(1);
[
  ...;
]
```

This script will be executed once only, and then the function will be reset to false (i.e. the script will not again execute unless the module, or containing state, is stopped and restarted).

```
If Watch(0, xValue, yValue);
[
  ...;
]
```

This script will be executed if either of the variables' values change by any amount, if they become invalid, or if their type changes.

Note: Note that the behavior will differ depending on whether you use this function in a script code module or in a tag expression. In script code, the function will be reset as described, and will wait for the next trigger to occur.
In a tag expression, this function will not be reset after triggering.

Related Information:
See: "Latching and Resetting Functions" in the VTScada Programmer's Guide

WatchArray

Description: Watches an array and returns true if any of its elements' types or values change.

Returns: Boolean

Usage: Steady State only.

Function Groups: Array, Variable

Related to: Change | Edge | Watch

Format: WatchArray(Start, ArrayElem, NumElem)

Parameters:

Start

Required. Any expression which evaluates to a false (0) or true (non-0) value. This will be the initial return value of the function.

ArrayElem

Required. Any numeric array element giving the starting element of the group of elements to watch. The subscript for the array may be any numeric expression. If processing a multidimensional array, the usual rules apply to decide which dimension should be examined.

NumElem

Required. Any numeric expression giving the number of array elements to watch. If NumElem extends past the upper bound of the lowest array dimension, this computation will "wrap-around" and resume at element 0, until N elements have been located.

Comments: This is a resettable function whose initial return value will be set by the first parameter. This parameter is ignored after the initial evaluation of the function.

Example:

```
If WatchArray(0, dataArray[0], 10);
[
```

```
  ...;
]
```

This script will be executed if any of the elements from 0 to 9 inclusive experience a change by any amount, if they become invalid, or if their type changes.

Related Information:

See: "Latching and Resetting Functions" in the VTScada Programmer's Guide

WatchForTagChanges

Description:	Watches for tag changes, either external or local
Returns:	Boolean
Usage:	Steady State only. See: Rules for Usage.
Function Groups:	Variable
Related to:	
Format:	WatchForTagChanges();
Parameters:	None
Comments:	To prevent multiple rapid external changes from causing heavy execution, a slight delay, during which there must be no changes, occurs between triggers

WCSubscribe

Description:	Working Copy Subscribe. After this function has been called, any configuration change will result in the specified callback subroutine being called.
Returns:	Nothing
Usage:	Script Only.
Function Groups:	Basic Module, Variable
Related to:	ReadConfiguration | ModifyConfiguration

Format: WCSubscribe(SubscriberObj[, CallbackModuleName])

Parameters:

SubscriberObj

A required object, in which the callback module will be called. Often, Self().

CallbackModuleName

An optional text value which is the name of the call-back object to be launched into the subscriber object. If invalid or not provided, there must be a submodule in the code, named Notify(), which will be used by WCSubscribe.

Comments: This function is commonly used in conjunction with ReadConfiguration(). The callback() will be notified when a particular file has changed and will trigger another module to call ReadConfiguration() to read the changes from that file.
"ChangedFiles" contains absolute file paths.

Example:

```
Init [
  If 1 Main;
  [
    \WCSubscribe(Self(), "ChangeNotification");
    ConfigChanged = 1                       { Cause initial read of
files };
  ]
]

Main [
  If ConfigChanged;
  [
    ConfigChanged = 0;
    \ReadConfiguration("ConfigReader");
  ]
]

<
{============================ \ChangeNotification
=======================}
{=============================================================================-
=====}
```

```
ChangeNotification
 (
  ChangedFiles                { Dictionary of changes files
};
)

Main [
  If 1;
  [
    IfThen(ChangedFiles["MyConfigFile.txt"],
      ConfigChanged = 1;
    );
    Return(Invalid);
  ]
]
>
```

WhileLoop

Description: Repeatedly executes a parameter while a condition is true.

Returns: Nothing

Usage: Script only.
May be used in optimized Tag Parameter Expressions.

Function Groups: Logic Control

Related to: Case | Cond | DoLoop | IfElse | IfThen

Format: WhileLoop(Condition, Function1, Function2, ...)

Parameters:

Condition

Required. An expression that will be evaluated to determine if the Function parameters should be executed. If it is true, the Function parameters will be executed and then Condition will be re-evaluated and the process repeated until Condition is either false or invalid.

Function1, Function2, ...

Required. Are any expressions which are to be executed while Condition is true. The Function parameters are executed in order.

Comments:	No other statement will execute as long as Condition is true. Care must be taken that the statement terminates since it has the potential of locking up the system if it does not exit.

Note: While the WhileLoop statement has its place in VTScada programming, it should be noted that speed can be enhanced by a factor of approximately 5 through the use of array processing functions (please refer to "Array Processing" for further details). Array functions are listed in "Array Functions".

Example:

```
i = 1;
...
If 1 Main;
[
  WhileLoop(i < 100 { Looping condition },
            arrayEven[i] = i * 2 { Store even numbers },
            arrayOdd[i] = i * 2 - 1 { Store odd numbers },
            i++);
]
```

This causes all even numbers from 1 to 198 to be stored in arrayEven and all odd numbers from 1 to 198 to be stored in arrayOdd.

WinButton

Description	Windows native button.
Returns	Integer
Usage	Steady State only.
Function Groups	Graphics, Window
Related to:	
Format:	WinButton(X0, Y0, X1, Y1, Style[, Text, FocusID, Font, ToggleVal, Bitmap, BitmapJustify])
Parameters	

X0, Y0, X1, Y1

Required. Any four numeric values, locating the edges of the button in the window. To ensure consistent sizing, these parameters should be set using constants. A commonly-seen example follows:

```
WinWd-2*BtnWd-2*Space, WinHt-Space, WinWd-
BtnWd-2*Space, WinHt-BtnHt-Space
```

Style

Required. Defines the button's appearance. A binary OR operation is done with a style number and the extra style bits shown in the following table.

If the style number is 0, the button will have a normal appearance. If the style number is 1, the button will be a toggling button. Further style refinements are as follows:

Value	Meaning
0x00000100	orient text to the left
0x00000200	orient text to the right
0x00000400	orient text to the top
0x00000800	orient text to the bottom
0x00002000	allow multiple lines of text on button

Text

An optional expression for text to display on the button.

FocusID

An optional parameter indicating the focus id. If Invalid or "0", no user interaction is permitted.

Font

An optional parameter specifying the font to use for text.

Note that underlining is not supported.

ToggleVal

An output value, whose meaning depends on the button style.

In a toggling button, ToggleVal is set to the toggle state of the button - 1 for pushed-in and 0 for not-pushed-in.

For this button style, ToggleVal can also be used to programmatically change the state of the button.

In a normal button, ToggleVal is purely an output parameter, indicating whether the button is currently being pressed.

Bitmap

An optional parameter specifying an image to use on the button. May be used with or without a value for the Text parameter. See example.

BitmapJustify

An optional numeric value, specifying how the image is aligned on the button. Possible values are as follows:

Value	Meaning
0	align image to the left (default)
1	align image to the right
2	align image to the top
3	align image to the bottom
4	align image to the center (not a valid option if text is also provided for the button)

Comments: None.

Example:

Plain button, with monitoring of the current toggle state.

```
If WinButton(232, 148, 412, 100,
             0                      { normal appearance },
             Concat("Press Me ", ButtonPresses) { label },
             1                  { focus id enables button },
             0                      { default system font },
             ToggleVal                    { monitor state});
[
  ButtonPresses++;
]
```

Button with image

```
MyImage = MakeBitmap("..\Bitmaps\warning symbol.bmp", 12, 9);
If WinButton(232, 148, 412, 100,
             0                      { normal appearance },
             Concat("Press Me ", ButtonPresses) { label },
             1                  { focus id enables button },
             0                      { default system font },
             Invalid                  { no ToggleVal used },
             MyImage     { image displayed in the button});
[
  ButtonPresses++;
]
```

WinComboCtrl

Note: Programmers are encouraged to use System\DropList instead.

Description: Windows native "combo" control. A "combo" control is an enhanced form of drop list. Displays a child window containing a Windows combo control.

Returns: Numeric. See comments.

Usage: Steady State only.

Function Groups: Graphics, Window

Related to: Droplist | Edit | WinEditCtrl

Format: WinComboCtrl(X0, Y0, X1, Y1, Style, Data, DataTrigger, Index [, FocusID, MaxCharacters, Font, BackgroundColor, ForegroundColor])

Parameters:

X0, Y0, X1, Y1

Required. Coordinates.

Style

Required. Comprised of a combination of bit values to yield the desired effects.
Bits 0 and 1 define mutually-exclusive styles of operation. They can be set to one of the following values. If neither is set, the result is a Listbox with selected item above it

Bit Number	Definition
0	Droplist. The current droplist selection is editable.
1	Droplist. The current droplist selection is not editable.

Bits 2 and 3 define input character handling. They can be set to one of the following values:

Bit Number	Definition
4	Input is converted to all uppercase.
8	Input is converted to all lowercase.

Bit 4 controls list sorting

Bit Number	Definition
4	List is presented to the user in alphanumerically sorted order.

Bits 5 and 8 are reserved.
Bit 9 Specifies that the size of the combo box is exactly the size specified by the application when it created the combo box. Normally, the system sizes a combo box so that it does not display partial items.

Bit Number	Definition
9	Enable application-defined geometry.

Data

Required. The one-dimensional array of data to be displayed.

Trigger

Required. Can be set to non-zero to cause control data to refresh. WinComboCtrl will set this parameter back to zero after the data has been refreshed.

Index

Required. The current selection index in the data array.

FocusID

An optional parameter indicating the focus id. If Invalid, negative, or zero, no user interaction is permitted.

This value is stored in a short. Values above 32767 will cause the control to be non-editable on VTScada Internet Clients.

MaxCharacters

An optional parameter indicating the maximum number of characters permitted for input.

Font

Required. An optional parameter indicating the font to use.

Note that underlining is not supported.

BackGroundColor

An optional numeric parameter specifying the background color. Uses an unsigned integer, therefore system colors with negative values may not be specified here.

ForegroundColor

An optional numeric parameter specifying the foreground color. Like BackgroundColor, may not be neg-

ative.

Comments: The return value for WinComboCtrl can be one of the following values:

Invalid - Nothing happened or there is a problem with the control.

0 Internal buffer changed

1 Selection made from list or Enter key pressed

2 Focus has been lost.

Data must be a one-dimensional array of text values. Index is a variable that receives the current selection array index, or the text value if the droplist has an editable selection and new text is entered that does not match any item in the array. If FocusID is Invalid or less than or equal to "0", the current selection cannot be modified.

Window

Description: Opens a new window, starts a module inside, and eventually returns the module's value.

Returns: Module Instance

Usage: Steady State only.

Function Groups: Basic Module, Window

Related to: SizeWindow | MoveWindow | VStatus | WindowClose | WindowOptions

Format: Window(Left, Top, ViewWidth, ViewHeight, VirtualWidth, VirtualHeight, Module, Style, Title, Color, Enable [, HelpFileName, HelpContext, enableDynamicPositioning])

Parameters:

Left

Required. Any numeric expression that gives the left coordinate of the new window. If the new window is a child window, Left is the user coordinate within the window of the calling module. If the new window is not a child window, Left is the number of pixels from the left of the screen.

Top

Required. Any numeric expression that gives the top coordinate of the new window. If the new window is a child window, Top is the user coordinate within the window of the calling module. If the new window is not a child window, Top is the number of pixels from the top of the screen.

ViewWidth

Required. Any numeric expression that gives the window width in pixels. The minimum width allowable is based on the operating system in use and the style options of the window (see the Comments section for more details).

ViewHeight

Required. Any numeric expression that gives the window height in pixels. This value must be greater than or equal to 0.

VirtualWidth

Required. Any numeric expression that gives the width inside the new window in user coordinates (which may be pixels). If VirtualWidth is larger than the client area specified, a horizontal scroll bar appears.

VirtualHeight

Required. Any numeric expression that gives the height inside the new window in user coordinates (which may be pixels). If VirtualHeight is larger than the client area specified, a vertical scroll bar appears.

Module

Required. Any expression that uses a module call. The module will start, and all graphics in the module will draw in the new window.

Style

Bit Number	Description
0	Enable system close button (if bit 1 is set)
1	Show title bar
2	Thick border, not resizable
3	Thin border, not resizable
4	Enable minimize button (if bit 1 is set)
5	Enable maximize button (if bit 1 is set)
6	Create window minimized
7	Create window maximized (full virtual size)
8	Disable scroll bars
9	Child window
10	Always on top
11	Reserved for future use
12	Modal window (like dialog box)
13	Use pixel coordinates in window (otherwise user coordinates)
14	Use Load statement to size window
15	Owned window
16	Initially inactive window
17	Invokes "automatic" alpha blending, where the window is set to be 50% translucent when inactive and opaque when active
18	Window is to be rendered as transparent, with whatever color is specified as the background color being the transparent color.
19	Indicates that a WindowClose statement

Title

Required. Any text expression that gives the window title.

Color

Required. Any numeric expression that gives the background color of the window's client area when the window opens.

If set to -1, then the resulting window will be transparent, as will anything drawn in black (RGB 0,0,0) upon it.

This parameter has no effect after the window has opened.

Any of the following may be used:

- a palette index VTScada Color Palette
- a Constants for System Colors
- an RGB string in the format, "<RRGGBB>"

Enable

Required. Any logical expression. If true, the window opens and Module starts. If false, the window is closed and Module is stopped.

HelpFileName

Optional file name of the help file to use if the user presses F1 while this window is the active window. If absent or invalid when the user presses F1, the parent window will be checked for a file name. This continues recursively until the top of the window tree is reached. If no help file name is found, the default help file is used. The default help file can be set through the Setup.ini variable WEBHelp, or by using the EnableHelp statement.

HelpContext

Optional help context. If absent or invalid, but the HelpFileName is valid, then the "finder" dialog for help

is displayed when the user presses F1. If valid and numeric, the help file is searched for a matching topic number and help is displayed for that topic.
If valid and textual, the help file is searched for an exact match on the text string among the topic index of the help file. If there is more than one text match, the index is positioned at the first partial string match.
If valid but a topic match is not found (neither textual nor numeric), the same action as an Invalid HelpContext is taken. If the HelpFileName parameter is Invalid, this parameter is ignored.

enableDynamicPositioning

An optional parameter that, when set to TRUE, will cause the window position to be updated if the window is a child window inside a GUITransform that moves, or if the LEFT or TOP parameters change. Set to FALSE by default.

RibbonName

Optional text string, identifying the ribbon. See comments.

RibbonState

Optional. A retained variable name, holding the persisted state of the ribbon.The persisted state is an XML snippet generated internally by the Microsoft ribbon API and is opaque to VTScada script code.
The retained variable is typically set in the destructor for the module instance the Window statement runs.

Comments: This function returns the value returned by Module.

When a window is maximized, its maximized size is

not based on the size of the screen, although for many commercial applications the two are the same, but rather, its full size is defined by the fifth and sixth parameters, namely virtual width and height. If the view area and the virtual area of the window have been defined to be the same, selecting the maximize button will appear to have no effect.

The minimum size of a window is based on the operating system under which the application is running, as well as the attributes of the window itself as defined by the Style parameter. The width need only be greater than or equal to 0. Window heights similarly need only be greater than or equal to 0. GUITransform functions may make use of VStatus to inquire about the boundaries of the window they are within.

A window can be turned into a drag and drop target by adding the callback modules, OLEDrag and OLEDrop. Further details can be found in the chapter, Create Windows & Use Graphics Functions.

If a ribbon is to be associated with this window, then the ribbon must be instantiated at the same time that the window is created, using the two optional parameters, RibbonName and RibbonState. Ribbons are compiled into resources, using Microsoft Visual Studio. The resources can then be linked directly into VTS.EXE (Trihedral use only) or into an external DLL. This allows multiple ribbons to be provided in separate DLLs. You can have more than one ribbon in VTS.EXE or a DLL.

RibbonName is of the form "dllname|ribbonname", being the name of the DLL, a vertical pipe symbol and the name of the ribbon resource within the DLL.

If the ribbon is compiled into VTS, the dllname and the pipe separator character are omitted.
If adding a ribbon to a window, and if there is a chance that the window will be maximized, you should use the window's virtual width and virtual height as follows:

```
Window(0,0,1024,768,
       VStatus(Self(),26), VStatus(Self(), 27) +
VSTatus(Self(), 23), ...
```

The virtual height in particular is important as otherwise there will be a gap at the bottom of the window (normally, eight pixels) that is the size of the frame height.

Example:

```
Window( 0, 0 { Upper left corner },
       800, 600 { View area },
       800, 600 { Virtual area },
       Graphics() { Start user graphics },
       {65432109876543210}
      0b00010000000110011, "Sample Window", 0, 1);
```

This statement will open a window, and run the module Graphics() in it.

Related Information:

See: "Create Windows & Use Graphics Functions" in the VTScada Programmer's Guide. A collection of best practices and related information.

WindowClose

Description:	Returns true if an attempt to close the window is made.
Returns:	Boolean
Usage:	Steady State only.
Function Groups:	Window
Related to:	VStatus | Window | WindowOptions
Format:	WindowClose(Object)

Parameters:

Object

Required. Any expression that returns the object value of any module instance drawn in the window.

Comments: If this function is active when an attempt to close a window is made, the window will not close, and the module(s) in that window will not stop. It is up to the calling module to close the window and stop the module(s) by switching to another state.

This function may be used to control what happens when a window is closed (or to prevent it from being closed). When the attempt is made to close the window, an action (or actions) can be triggered. For example, this might write some data to file. Then a variable could be set (or a value could be returned by the module) which signals the calling module to switch states, which closes the window and stops the module(s) inside.

Note: It is possible for a WindowClose statement to be associated with a window, but to not yet be running when the attempt to close the window is made. If this happens the slay of the module running in the window will continue up the module tree until a running WindowClose is found, or the top of the tree is reached. This behavior may be undesirable. The scenario can be avoided by setting bit 19 of the Style option in the Window statement. Doing so will warn VTScada that a WindowClose statement will be associated with this window.

Example:

```
If ZButton(110, 240, 200, 270, "CANCEL", 1) ||
          WindowClose(Self()) CloseEverything;
```

This statement causes a state change to CloseEverything if the Cancel button is selected, or if the toaster bar on the window is selected.

WindowOptions

Description: Alters the options on a window once it has been opened.

Returns: Nothing

Usage: Script or Steady State

Function Groups: Window

Related to: MoveWindow | SizeWindow | Window | WindowClose

Format: WindowOptions(Object, Option, OptValue);
Or
WindowOptions(Object, 15, Width, Height);

Parameters:

Object

Required. Any expression that returns the object value of any module instance drawn in the window.

Option

Option	Attribute to alter
0	Visibility of the window
1	Horizontal scroll line step size
2	Horizontal scroll page step size
3	Vertical scroll line step size
4	Vertical scroll page step size
5	Set Horizontal Scroll position
6	Set Vertical Scroll position
7	Locator reporting rate from the VIC is accelerated. Typically this would be used when fast, dynamic feedback of mouse movement from the VIC is required for a rapid, interactive response. CAUTION: Unnecessary use of this option can seriously degrade the VTScada window update rate on the VIC.
8	Locator reporting rate from the VIC is returned to normal. Locator position reports are only sent when the VIC considers them significant. This is the default mode of operation.
9	Sets the level of alpha-blending (window translucency) of the window. OptValue determines the level of alpha-blending from 0 (completely transparent) to 255 (completely opaque). An OptValue of 128 would give 50 percent translucency.
10	Set background color of tooltips. Must

OptValue

Required. Any numeric expression that specifies the value to set the given option to.
For an Option of 0 (visibility), OptValue produces the following effects:

OptValue	Effect
0	Activate and display in current position
1	Bring to foreground without activating
2	Maximize and activate
3	Minimize. Deactivate if active
4	Hide
5	Similar to OptValue 0, but the graphic editing window in-use is not modified.

Width

Required. Used when the Option value is 15. Sets the minimum window width

Height

Required. Must be present and valid when the Option value is 15. Sets the minimum window height.

Comments: This statement will alter the attributes of the window as it is running. The values that the given options can be set to are from -32763 to 32763.
When the option value is set to 15, the function requires four parameters, all of which must be valid. The last two set the minimum width and height

Example:

```
If MatchKeys(2, "I");
[
```

```
  WindowOptions(Self(), 0, 3);
]
```

If an "I" is pressed on the keyboard, the window that this module is running in will be minimized.

```
If ZButton(10, 50, 110, 80, "SetSize", 2, \System\DefFont);
  [
    WindowOptions(Self(), 15, 200, 150);
  ]
  If ZButton(10, 90, 110, 120, "ClearSize", 3, \System\DefFont);
  [
    WindowOptions(Self(), 15, 0, 0);
  ]
```

WindowsLogon

Description: Authentication request to Windows Authentication services.

Returns: Boolean to indicate success or failure

Usage: Steady State only.

Function Groups: Security

Related to:

Format: WindowsLogon(Username, Password)

Parameters:

Username

Required. The account name to authenticate.

Password

Required. The password, required to authenticate the given account.

Comments: This function should not be used for VTScada-based authentication.
Note that this function may take some time to complete and will block the caller until it does. Other threads in the VTScada system will continue to run while this function is executing.

WindowSnapshot

Description: Creates an image file containing a screen capture of the specified window.

Returns: Boolean indicating success (TRUE) or failure of the operation.

Usage: Script Only.

Function Groups: Window

Related to:

Format: WindowSnapshot(Filename, WindowObj, MimeType[, Left, Top, Right, Bottom]);

Parameters:

Filename

Required. Any expression that is a string, or an array of strings containing the full filename(s) to which the screen capture should be saved.

WindowObj

Required. An object (often, "Self()") specifying the window to be captured.

MimeType

Required. Any expression that is a string, or an array of strings specifying the mime type(s) to be used for the image.

Left, Top, Right, Bottom

Optional. Short integer values, providing the bounding area of the region to be captured within the window. If any one of these parameters is provided, all four must be provided. If not provided, a snapshot will be taken of the entire window.

Comments: Available mime types include image/bmp, image/jpeg, image/gif, image/tiff, image/png.

Note: Runs asynchronously from a VTScada Internet Client. In this case, the return value is meaningless.

> Note: Within an Anywhere Client session, this function does nothing.

Example:

```
[
  Mimetypes                   { Set of mime types we can snapshot to
};
  Filetypes                   { Corresponding file types (extensions)
};
  SelectedMimetype            { Work var - iterator through above
arrays        };
  Constant #NUM_TYPES = 5     { Number of entries in the above arrays
};
]
Init [
  {***** Allow enough time for the window to paint *****}
  If Timeout(1, 1) TakeSnapshots;
  [
    Mimetypes = New(#NUM_TYPES);
    Mimetypes[0] = "image/bmp";
    Mimetypes[1] = "image/jpeg";
    Mimetypes[2] = "image/gif";
    Mimetypes[3] = "image/tiff";
    Mimetypes[4] = "image/png";
    Filetypes = New(#NUM_TYPES);
    Filetypes[0] = "bmp";
    Filetypes[1] = "jpg";
    Filetypes[2] = "gif";
    Filetypes[3] = "tiff";
    Filetypes[4] = "png";
    SelectedMimetype = 0;
  ]
]

TakeSnapshots [
  {***** Snapshot the window for each mime type *****}
  If SelectedMimeType < #NUM_TYPES;
  [
    WindowSnapshot(Concat(PathString, "\Lighthouse.", FileTypes[Selec-
tedMimetype]),
                   Self(),
                   Mimetypes[SelectedMimetype]);
    SelectedMimetype++;
  ]
```

WinEditCtrl

Description: Windows native edit control. This function returns a value

indicating the status of an edit field.

Returns: Numeric

Usage: Steady State only.

Function Groups: Graphics

Related to: Droplist | Edit | WinComboCtrl | ZEditField

Format: WinEditCtrl(X0, Y0, X1, Y1, Style, Data [, FocusID, MaxCharacters, Font, BackgroundColor, ForegroundColor])

Parameters:

X0, Y0, X1, Y1

Required. Coordinates.

Style

Optional. Default 0. Comprised of a combination of bit values to yield the desired effects. Note that some combinations should not be used as they could be mutually exclusive; for example converting input to all uppercase and to all lowercase (bits 2 and 3 both set).
Bits 0 and 1 are reserved for bit compatibility with WinComboCtrl, and should be set to "0".
Bits 2, 3 and 4 define input character handling. It is reasonable to set bit 2 and 4 or 3 and 4, but not bits 2 and 3. If not set, input is used exactly as typed.

Bit Number	Definition
2	Input is converted to all uppercase.
3	Input is converted to all lowercase.
4	Input is masked. Any characters typed will appear as asterisks. (use-

ful for password fields)

Bit 5 controls multi-line edit controls.

Bit Number	Definition
5	Multiline editing. When set, this bit causes a typed Enter key to be interpreted as "move to the start of the next line". Text that contains carriage-return & line-feed characters has a line break inserted at each set.

Bit 6 forces the background color when the control is disabled, rather than allowing it to gray as it should.

Bits 8 and 9 are reserved.

Data

Required. The data to be displayed and possibly edited.

FocusID

An optional parameter specifying the focus id. If Invalid, negative, or zero, no user interaction is permitted. This value is held in a short. Values above 32767 will cause the control to be non-editable on VTScada Internet Clients.

MaxCharacters

An optional parameter specifying the maximum number of characters permitted for input. Defaults to 32767.

Note that this property is not obeyed in a VTScada Internet Client.

Font

An optional parameter specifying the font to use. Note that underlining is not supported.

BackGroundColor

An optional numeric parameter specifying the background color. Uses an unsigned integer, therefore system colors with negative values may not be specified here. May be any of:

- a palette VTScada Color Palette
- a Constants for System Colors
- an RGB string in the format, "<RRGGBB>"

ForegroundColor

An optional numeric parameter specifying the foreground color. Like BackgroundColor, may not be negative.

Comments: The return value for WinEditCtrl can be one of the following:

Return Value	Meaning
Invalid	Nothing happened or there is a problem with the control
0	Internal buffer changed or selection made from list.
1	<CR> (Enter or Return pressed). Only applies to single-line edit controls.
2	Focus has been lost.

If FocusID is Invalid or less than or equal to "0", the current selection cannot be modified.

WinLocSwitch

Description: Returns the current status of the locator (mouse) buttons in a certain window and its ancestors.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Locator, Window

Related to: Click | LocSwitch | SetXLoc | SetYLoc | Target | WinXLoc | WinYLoc | XLoc | YLoc

Format: WinLocSwitch(Object)

Parameters:

Object

Required. Any expression that returns the object value of any module instance drawn in the window.

Comments: If the mouse isn't over the specified window or one of its ancestors, the function returns the mouse button status for the last time it was over the window (or ancestor). If the locator is not installed, the function returns 0. Otherwise, the return value has the following significance:

Return Value	Mouse Button(s)	No. of Clicks
0	No buttons	-
1	Right button	Single
2	Middle button	Single
3	Right and middle buttons	Single
4	Left button	Single
5	Left and right buttons	Single
6	Left and middle buttons	Single
7	All three buttons	Single
8	No buttons	-
9	Right button	Double
10	Middle button	Double
11	Right and middle buttons	Double
12	Left button	Double
13	Left and right buttons	Double
14	Left and middle buttons	Double
15	All three buttons	Double

Note: It cannot be over-emphasized that this function looks at the status of the mouse over not only the window indicated by Object, but all ancestral windows of Object as well. That is to say, any of its children, grandchildren, parents, grandparents, etc. For example, a WinLocSwitch statement in a certain module, we'll call it ModA, and a WinLocSwitch statement in a child module of ModA, call it ModB, will

both return a value of 4 if the left mouse button is pressed over either one of them. If this action is not what is required for your application, the LocSwitch function may be more appropriate, since it will return a value of 4 only if the left mouse button is pressed while over the same window containing the module with the LocSwitch statement in it. From the previous example, this means that ModA's function will return the expected value only when over the window containing that module, not when the mouse is over ModB's window, and vice versa.

Example:

```
If WinLocSwitch(Cond(CurrentWindow() != secondaryWin,
                CurrentWindow(),
                Invalid())) == 4 TestDraw;
[
  ...
]
```

This statement will check the status of the left mouse button if its current window is not the one called secondaryWin. If there has been a change over any of the windows but secondary window, the script will execute and a change of state to TestDraw will occur. At some point prior to this, secondaryWin would have been set to a module instance inside the desired window.

WinMatchKeys

Description:	Returns true if the specified keyboard keys have been pressed in the sequence given, in another window.
Returns:	Boolean
Usage:	Steady State only. See: Rules for Usage.
Function Groups:	Keyboard
Related to:	MatchKeys
Format:	WinMatchKeys(Object, Enable, Keys)
Parameters:	

Object

Required. Any expression which gives the object value of any module instance which is running in the window.

Enable

Required. Any numeric expression giving an enable for the function. Testing of keyboard input is enabled when this parameter is true (not 0). If this parameter is false(0), the function's value is false(0).
In addition, the Enable parameter controls the type of comparison done. If the Enable is 1, a case-sensitive match is made. If the Enable is 2, then the match is not case-sensitive.
(Any non-zero value other than 2 will cause a case-sensitive match. The use of 1 and 2 is recommended for clarity.)

Keys

Required. A text expression giving the key sequence to test for. The case of individual letters may be significant, depending on the Enable parameter.

Comments:

The Object is used to determine which window to watch for the keystrokes in.
The Enable is a status expression controlling the comparison. The comparison starts once the Enable becomes true. If the Enable becomes false, the function's value becomes false and the comparison starts at the beginning of the Keys string again once the Enable becomes true. This feature is useful for resetting the WinMatchKeys function once an action using the function's result has been performed.

The WinMatchKeys function is also reset automatically when it occurs in an action trigger that becomes true.
The function's result is automatically set to false(0) when the state containing the function is entered. Once the function becomes true, it remains true as long as the state does not change and the Enable remains true.
Any key sequence may be used for the Keys parameters including the function keys. Note that the WinMatchKeys function is case sensitive (upper and lower case letters are treated as different characters) when the Enable is an odd number. Often only one key is included in the Keys string. Several keys may be used in the Keys string and function as a password. The keys typed are not displayed on the screen by this function. Several WinMatchKeys functions may be active at any time, each comparing the keyboard input against their own Keys parameter.

Example:

```
If WinMatchKeys(secondaryWin, 2, "go");
[
  ...
]
```

This statement will wait until the window designated by secondaryWin is active and then will further wait until the keys "go" (in any case, upper or lower) is entered at the keyboard. At that point the function will return true and the script will execute.

Related Information:

See: "Latching and Resetting Functions" in the VTScada Programmer's Guide

WinShiftKeys

Note: Deprecated. Works only on VTScada servers, not on Internet clients. Use WinMatchKeys for new code.

Description: Returns a value which contains the current status of the

Shift, Control and Alt control keys.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Keyboard

Related to: WinMatchKeys

Format: WinShiftKeys(Object)

Parameters:

Object

Required. Any expression that gives the object value of any module instance which is drawn in the window.

Com-ments: The return value will be a sum of the individual key values (i.e. each key has a bit which when set indicates that the key is currently pressed)

Return Value	Bit No.	Key
1	0	Shift
2	1	Control
4	2	Alt
8	3	Caps lock (locked on)
16	4	Num lock (locked on)
32	5	Scroll lock (locked on)
64	6	Left arrow
128	7	Down arrow
256	8	Right arrow
512	9	Up arrow
1024	10	Page down
2048	11	Page up

WinTooltipCtrl

Description: Windows native "tooltip" control. A "tooltip" is a pop-up text window that provides operational hints to users when the mouse pointer is rested over a tool or object.

Returns: Invalid

Usage: Steady State only.

Function Groups: Graphics, Locator

Related to: WindowOptions

Format: WinTooltipCtrl(X0, Y0, X1, Y1, Style, Text [, Title, Icon-Index, Enable, Font])

Parameters:

X0, Y0, X1, and Y1

Required. Coordinates of a rectangular screen area. When the mouse enters that area (the "hit area"), and stops moving for a time, the pop-up tooltip will be displayed.

Style

Required. One or more of the following bit flags, used to control the style of the tooltip:

- 1 – If set, the tooltip will respond to the mouse in the hit area regardless as to whether the containing window is the active window. If clear (default), the tooltip only responds if it is the active window.
- 32 – If set, fading of the tooltip is disabled. If clear (default), the tooltip fades in and (under some circumstances) out.
- 64 – If set, the tooltip is displayed as a balloon. If clear (default), the tooltip is displayed as a rectangular window.
- 128 – If set, the tooltip is displayed in-place (i.e.

over the top of the hit area). This is most useful when the hit area is occupied by clipped text and the tooltip contains the full text, so that hovering over the clipped text shows the full text in a tool-tip window positioned over the top of the clipped text. Note: \System\ListBox uses this ability.

Text

Required. Any expression yielding a simple text value or a 1-dimensional array of text values. If this yields a simple text value, the text value is displayed in the tool-tip control, interpreting any carriage return/line feed pairs (CRLF) as a line break, causing what follows to be displayed on the following line. If this yields an array, each element of the array is displayed in a new line of the tooltip control.

Title

An optional parameter that can be any expression yielding a simple text value. The text value is displayed in the tooltip control as a title (emboldened at the top of the tooltip).

IconIndex

Selects an optional icon to be displayed to the left of the title:

Invalid or 0 - No Icon

1 - Information Icon

2 - Warning Icon

3 - Error Icon

Note that while the operating system may define other icons for other values of this parameter, these are not defined at the present, and are hence subject to change.

Enable

An optional parameter that if non-zero enables the

popping-up of the tooltip. If Invalid, the default is true (i.e. pop-up enabled).

Font

An optional parameter that can be set to a font value for the font to be used by the tooltip. If Invalid, the default system font is used.
Note that underlining is not supported.

Comments: This statement returns Invalid.
The background color, text color, and delay timings are set on a per-window basis using the WindowOptions statement.

Note: Detailed information about native Windows tooltip support is provided in "Common Tasks: Native Windows Tooltip Support".

WinXLoc

Description: Returns the X coordinate of the locator (mouse) for a window.

Returns: Numeric

Usage: Script or steady state.

Related to: WinLocSwitch | WinYLoc | XLoc | YLoc | GUITransform

Format: WinXLoc(Object)

Function Groups: Graphics, Locator, Window

Parameters:

Object

Required. Any expression that gives the object value of any module instance which is drawn in the window.

Comments: Returns the location of the mouse for a particular window. If the mouse is not over that window, it retains the value from the last time that the mouse was over that window. If the window that the mouse

is over is a child window, a WinXLoc/WinYLoc statement in the parent will register the same values as one in the child. That is, the coordinate position in the child window. Similarly, when the mouse is moved over the parent, the values for WinXLoc/WinYLoc in both the parent and child windows will be the same. That is, the location of the mouse in the parent window.
Note: This function is disabled when using a GUITransform as a GUIStrectch.

Example:

```
If Change(WinXLoc(myWin), 0) || Change(WinYLoc(myWin), 0);
[
  ...
]
```

This statement checks for any movement of the mouse in the window designated by myWin and executes the script when it occurs.

WinYLoc

Description: Returns the Y coordinate of the locator (mouse) in a window.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Graphics, Locator, Window

Related to: WinLocSwitch | WinXLoc | XLoc | YLoc | GUITransform

Format: WinYLoc(Object)

Parameters:

Object

Required. Any expression that gives the object value of any module instance which is drawn in

the window.
Note: This function is disabled when using a GUITransform as a GUIStrectch.

Example:

```
ZBar(10, 500, 100, WinYLoc(myWin), 14);
```

This statement draws a yellow bar on the left side of the window whose top follows the movement of the mouse in the window designated by myWin.

WKSList

Description: Generates a list of sub-paths from the query returned by the WKSPath function.

Returns: Array

Usage: Script Only.

Function Groups: Network

Related to: WKSPath | WKSStatus

Format: WKSList(QueryPath)

Parameters:

QueryPath

Required. A query path as generated by WKSPath. Due to the complexity of the included symbols and formatting rules, this path should not be hand-coded.

Comments: Returns an array of sub-paths that are a refinement of the query path returned from WKSPath. If the query path provided is a broad category, the resulting array may be quite large.
Returns a Windows™ PDH error code upon failure. This is a long integer containing a 32-bit code.

WKSPath

Description: Given set of path components, generates a query path for use in the WKSStatus command.

Returns: Text

Usage: Script Only.

Function Groups: Hardware and Software, Network

Related to: WKSList | WKSStatus | WKStaInfo

Format: WKSPath(MachineName, ObjectName[, InstanceName, ParentInstance, InstanceIndex, CounterName)

Parameters:

MachineName

Required. The name of the computer to query. Can be obtained dynamically using the function call, WkStaInfo(0);

ObjectName

Required. The system object to query. (corresponds to the objects in the top list in the provided image). "Processor" is an example.

InstanceName

The name of the instance of the system object to query. The wildcard, * may be used to query all instances.

ParentInstance

The name of the parent instance of the system object. The wildcard, * may be used to query all parent instances.

InstanceIndex

An integer assigned to the system object, if unnamed.

CounterName

The name of the data item to query from the system object. The wildcard, * may be used to query all sub

items of the system object.

Comments: If the optional parameters are to be included but not specifically set, then wildcard values must be provided. For text parameters, use the asterisk *, and for numeric parameters, use -1.
The output from WKSPath will usually be passed to WKSList for further processing, then to WKSStatus to use to query workstation status. The parameters to this function and the resulting query path are best understood in the context of the Windows™ Performance Monitor.
Returns a Windows™ PDH error code upon failure. This is a long integer containing a 32-bit code.

Example:

```
  If 1 Main;
  [
    MachineName    = WkStaInfo(0);  { Set up some defaults }
    ObjectName     = "Processor";
    InstanceName   = "*";      { These last four match the
                                 native defaults of the WKSPath func-
tion }
    ParentInstance = "*";
    InstanceIndex  = "0";
    CounterName    = "*";
  ]
]

Main [
  If ZButton(10, 70, 950, 50, "Create Query", 7);
  [   { Converts a set of parameters into a query string }
    Query = WKSPath(MachineName, ObjectName, InstanceName,
                    ParentInstance, InstanceIndex, CounterName);
  ]
```

WKSStatus

Description: Sends a query to the Windows™ Performance Monitor interface (see image in WKSPath) and returns the result as a query handle.

Returns: Handle

Usage: Script Only.

Function Groups: Hardware and Software, Network

Related to:

Format: WKSStatus(Query, OperationType)

Parameters:

Query

Required. The query to be sent to the Performance Monitor. The query path should be built using WKSPath, possibly refined by being passed through WKSList.

OperationType

Required. Controls the operation according to the following values:

OperationType	Action
0	Get value
1	Get error value.
2	Generate handle (QueryHandle must be a path)
3	Clear handle

Comments: Returns a number of type double as a result of the query provided. Upon error, the result will be invalid. The operation type can be set to 1 in order to obtain the error code. Note that some queries must be checked multiple times before a result will be produced, therefore not all error codes are critical.

Related Functions:

WKSPath | WKSList

Example:

```
handle = WKSStatus("\\MyPC\Processor(_Total)\% Processor Time", 2);
         { get a handle that will query total processor time }
```

```
IfThen(status = WKSStatus(handle, 1), { test for errors }
   … error …
    { error code zero means things are okay,
        everything else is a fault code }
);
value = WKSStatus(handle, 0);
     { make a priming read - some queries require two reads before
        producing data }
   … wait one second …
     { let some time pass before the second read }
value = WKSStatus(handle, 0);  { read some data }
IfThen(status = WKSStatus(handle, 1),
     { before using the data make sure there weren't any problems }
   … error …
);
WKSStatus(handle, 3);
     { close the handle when done, note that the same handle can be
       reused as long as needed. }
     { A new handle is not necessary for each read }
```

WKStaInfo

Description: Returns the characteristic information about this work-station.

Returns: Text

Usage: Script only.
May be used in optimized Tag Parameter Expressions.

Function Groups: Network, Hardware and Software

Related to: VStatus | Platform | BuffToHex

Format: WKStaInfo(Option)

Parameters:

Option

Required. Selects the characteristic to return, as shown in the following table

Option	Return characteristic
0	Workstation name
1	User name
2	Domain name
3	Unique Machine Identifier
4	Returns TRUE if the local machine is configured to resolve names using DNS.
5	Returns the domain name of the computer.

Comments: This function will only return a valid value if the machine has a workstation name assigned to it (i.e. if network services have been installed). If network services have not been installed it will return invalid.
Option 3, the unique machine identifier, returns a 6-byte binary buffer. Use the function \System\BuffToHex on the returned value if you would like to use the MachineID as a Hex string.

Note: VTScada relies on a Windows function call to obtain the NetBios name of workstations. That function will truncate names that are longer than 15 characters. You are advised to limit workstation names to be 15 characters or less.

Write

(VTSDriver Library)

Description: Used by a tag to create a write request to a driver address.

Returns: Object value of underlying write module.

Usage: Script Only.

Function Groups: String and Buffer, Stream and Socket

Related to: AddRead

Format: …\Driver\Write(Address, N, Val[, DType, TagName, Success])

Parameters:

Address

Required. The starting address to write the data to.

N

Required. The number of elements to write.

Val

Required. The data to be written. If Val is a single value or a statically-declared array, it should be preceded by a pointer reference (&). Do not use the pointer reference if passing a dynamically-declared array of values.

DType

No longer used. Was: the data type to be written to the I/O device.

TagName

The name of the tag that is writing the data.

Success

Pointer to a Boolean, used to pass success/fail information.

Comments: Allows the writing of a specific address on demand. The resulting data will be sent only to the requesting machine. Will not work for client writes via a server.
The object value of the underlying read module is returned from the function. When the write finishes, the returned object's value will go to Invalid, signaling the end of the write operation.

Example:

{ Simple variable example - writes a value of 12 to address 49500 }

```
WriteData = 12;
\Root\Driver\Write(49500, 1, &WriteData, Invalid(), Invalid(), &Suc-
cess);
```

{ Multi-variable (array) example - writes 12, 14, 30123 to addresses 40001, 40002, & 40003 respectively }

```
WriteData = New(3);
WriteData [0] = 12;
WriteData [1] = 14;
WriteData [2] = 30123;
\Root\Driver\Write(40001, 3, WriteData, Invalid(), Invalid(), &Suc-
cess);
```

WriteHistory

(Historian Manager Library)

Description: Interface to write tag history.

Returns: Numeric

Usage: Script Only.

Function Groups: Log

Related to: GetTagHistory

Format: \HistorianManager\WriteHistory(TagObj[, TimeStamp, Data, HistorianObj, BroadcastMode])

Parameters:

TagObj

Required. The tag instance for which the data is to be written

TimeStamp

Optional UTC timestamp.
If invalid, a timestamp will be generated using the current time.
If the number of records to be written is 1 then this should be a simple value.
If several records are to be written, this should be a

simple array having a size that matches the number of records.

Data

Optional data values to record.
If invalid, current values from TagObj will be used.
Otherwise, this may be:

- A simple value to log a single variable.
- A simple array, the size of which must match the number of records to write if the number of variables for each record is 1. Otherwise, this must match the number of logged variables for each record.
- An array of arrays – to be used only if the number of logged variables for each record (NumLogVars) is > 1 and the number of records to record (NumRecords) is > 1. The containing array size must match NumLog Vars and the nested array size must match NumRecords. See examples.

HistorianObj

An optional instance of the Historian tag used to log this value. If invalid, TagObj\HistorianName will be used.

BroadcastMode

An optional Boolean value. If TRUE, then the history is automatically relayed to all Historian-potential servers. Defaults to FALSE.

Comments: In order to ensure that the history is successfully written, this function must be called with the same data on all Historian-potential servers, (which can be done by calling it on all workstations,) unless BroadcastMode is TRUE, in which case it need only be called on exactly one workstation (any work-

station).

Possible return values are as follows:

-1 if the number of timestamps passed in does not match the number of data entries passed in.

0 otherwise. This does not indicate that the data was written, just that the above condition on the parameters held true.

Examples:

Simple case:

```
\HistorianManager\WriteHistory(TagObj, TimestampInUTC, Value);
```

More complex use case:

Given a tag with two logged variables, defined as:

```
Value(5)
Comment(6)
```

Perhaps you would like to log 3 records for each of them.

```
NumLoggedVars = 2;
NumRecords    = 3;
```

```
TimestampsinUTC      = New(NumRecords);
TimestampsinUTC[0] = CurrentTime(1);
TimestampsinUTC[1] = CurrentTime(1) + 1;
TimestampsinUTC[2] = CurrentTime(1) + 2;
```

```
ValueArray    = New(NumRecords);
ValueArray[0] = 10;
ValueArray[1] = 20;
ValueArray[2] = 30;
```

```
CommentArray    = New(NumRecords);
CommentArray[0] = "Comment1";
CommentArray[1] = "Comment2";
CommentArray[2] = "Comment3";
```

```
DataArray     = New(NumLoggedVars);
DataArray[0] = ValueArray;
DataArray[1] = CommentArray;
```

```
\HistorianManager\WriteHistory(TagObj, TimestampsInUTC, DataArray);
```

Note that, when building the array of arrays, the order is important. The values should correspond to the variable declared alphabetically by variable class type. Thus, Value(5) comes before Comment(6). See: Data Logged or Trended Variables in Tag Modules.

WriteINI

Note: Access to configuration files is not reliable unless the caller holds the working copy lock. Acquiring the lock is a steady-state only operation, and therefore legacy operations that used script-mode access to these files are deprecated or no longer supported (see comments)

(System Library)

Description: This subroutine writes a variable's value to a configuration file or a buffer containing one and returns its error code. Will not access .Startup or .Dynamic files.

Returns: Numeric

Usage: Script Only.

Function Groups: File I/O

Related to: CheckFileExist | CheckPathExist | ReadINI | ReadSectINI | WriteSectINI

Format: \System\WriteINI(File, Section, VarName, Value [, UseBuff])

Parameters:

File

Required. Any text expression giving the absolute path and file name of the Settings file or a pointer to the buffer containing its contents, depending on parameter UseBuff.

Section

Required. Any text expression giving the name of the section in the file. This should not include the square brackets delimiting the section.

VarName

Required. Any text expression giving the name of the variable for which the value is to be set.

Value

Required. Any giving the value to be assigned to variable VarName.

UseBuff

An optional parameter that is any logical expression. If true (non-0) the value of File must be a pointer to a buffer, if false (0) it is a file that is to be used. The default used if this parameter is omitted is false.

Comments:

For developers the lock means that access to VTScada working copy files, both reading and writing, should not be done without having the lock. The lock is across all applications and system layer VTScada code. The lock prevents two different piece of code from changing the same code such that one piece of code sees inconsistent data while the other code is in the middle of changing it.
This subroutine was a member of the System Library, and must therefore be prefaced by \System\, as shown in the "Format" section. If you are developing a script application, use "System\..." rather than "\System\..." in the function call.
The subroutine returns true (1) if the write was successful and invalid otherwise. If the Settings file, the section, or the variable does not exist, they will be created.
Searches performed by this function are case insensitive.
Alignment of equal signs in the file is preserved.

Example:

```
If 1 Main;
[
  System\WriteINI("C:\VTScada\Setup.ini"){filename},
                "System " { Name of section },
                 "OrderlyShutdown" { Name of variable },
                 10 { Value of variable },
                 0 { Write file format });
]
```

This assigns a value of 10 to the variable OrderlyShutdown in the System section of the Setup.INI file.

WriteINIProperties

Description: Writes properties to the local layer's various settings files in one operation.

Returns: Nothing

Usage: Script Only.

Function Groups: Configuration Management

Related to: WritePropertiesFile |

Format: Layer\WriteINIProperties(INIProperties[, ExternalLock])

Parameters:

INIProperties

Required. A dictionary containing the properties and values to be written. This dictionary must have the same structure as that returned by ReadINIProperties.

ExternalLock

Optional Boolean. Set to TRUE if you do not want to acquire and release the lock. Defaults to FALSE.

Comments: This function is the opposite of ReadINIProperties.
In nearly all cases, it is recommended that WritePropertiesFile be used in place of this function.

Examples:
none

WriteLock

(RPC Manager Library)

Description: This subroutine attempts to require a Write lock for the specified service. Subroutine call only.

Returns: Nothing

Usage: Script Only.

Function Groups: Network

Related to: ReadLock

Format: \RPCManager\WriteLock(ActivePtr, Service [, OptGUID]);

Parameters:

ActivePtr

Required. A reference to a variable that will be set to "1" when the Read lock is obtained.

Service

Required. The name by which the service is known.

OptGUID

An optional parameter indicating the GUID of the application in which the service instance is located. The default is the application to which the caller belongs.

Comments: This subroutine is a member of the RPC Manager's Library, and must therefore be prefaced by \RPCManager\, as shown in the "Format" section. If the application you are developing is a script application, the subroutine call must be prefaced by System\RPCManager\, and the System variable must be declared in AppRoot.SRC.

WritePropertiesFile

(System Library)

Description: Write a single Settings file according to the properties in an INIFile structure.
Replaces WriteINI and WriteSectINI

Returns: Boolean indicating success or failure.

Usage: Script Only.

Function Groups: Configuration Management, File I/O

Related to: ReadPropertiesFile | GetINIProperty | SetINIProperty

Format: \System\WritePropertiesFile(INIData, TargetDirectory[, IsBuffer])

Parameters:

INIData

Required. An INIFile data structure, containing the file name to write to and the application properties to be written. Created as follows... IniData = \System\INIFiles();
See the Comments section for a description of the INIFile data structure.

TargetDirectory

A text expression providing a directory name to be concatenated in front of the FileName provided by the INIData parameter. If left blank, a base path to the VTScada install directory will be appended to the what is in the FileName member of the INIFile structure.

IsBuffer

An optional logical expression. Set TRUE if the TargetDirectory parameter is a buffer, which will receive the output of the function. Defaults to FALSE (0).

Comments: Many properties can be modified with a single call to WritePropertiesFile.

The INIFile structure is as follows:

```
INIFiles Struct [
  FileName    { File name to the settings file.
Path may be included,
                but is better specified in the
TargetDirectory
                parameter.
};
  OEM         { TRUE if an OEM layer file
};
  Workstation { Name of the workstation or
invalid if global          };
  Layer       { Instance of application layer
owning the file         };
  Dynamic     { TRUE if a dynamic property
};
```

```
  Sections    { Dictionary of sections each ele-
ment of which
                is an array of Property struc-
tures        };
  Changed     { User sets to true if the file
has been changed,
                initialized to false
};
 ]
```

The INIProperty structure is…

```
INIProperty Struct [
    Name        { Variable name in the .star-
tup/.dynamic file  };
    Value       { Simple value
};
    Comment     { Text comment if present in the
file          };
    Hidden      { TRUE if not visible in Edit
Properties GUI   };
  ];
```

Note that if your intention is to write to a configuration file, this function should be called from within a ReadConfiguration callback or a ModifyConfiguration callback.

Example:

A \System\INIProperty() structure is used in the INIFIles\Sections

```
{ Setup the INIFiles Struct }
INIData = \System\INIFiles();
INIData\FileName    = Concat(ProfileName, ".ini");
INIData\Workstation = Invalid;
INIData\Dynamic     = 0;
INIData\Sections    = Dictionary();
INIData\Changed     = 0;
{ Now write to the .ini file }
\System\WritePropertiesFile(INIData, "MyProject\Profiles\");
```

WriteSectINI

(System Library)

Description: This subroutine writes an entire section to a configuration file or a buffer containing one and returns its error code.
Will not access .Startup or .Dynamic files.
Access to configuration files is not reliable unless the caller

holds the working copy lock. Acquiring the lock is a steady-state only operation, and therefore similar legacy operations that used script-mode access to these files are deprecated or no longer supported (see comments)

Returns: Numeric

Usage: Script Only.

Function Groups: File I/O

Related to: CheckFileExist | CheckPathExist | Edit | Folder | ReadINI | ReadSectINI | WriteINI

Format: \System\WriteSectINI(File, Section, VarList [, UseBuff])

Parameters:

File

Required. Any text expression giving the absolute path and file name of the Settings file or the name of the buffer containing its contents.

Section

Required. Any text expression giving the name of the section in the file. This should not include the square brackets delimiting the section.

VarList

Required. A 2 dimensional array containing the variables and their values to write to the file. The first row, VarList[0][N], contains the variable names, while the second row, VarList[1][N] contains their values.

UseBuff

An optional parameter that is any logical expression. If true (non-0) the value of File must be a pointer to a buffer, if false (0) it is a file that is to be used. The default used if this parameter is omitted is false.

Comments: WriteSectINI is a subroutine, and as such it needs to return

quickly. It therefore cannot wait for a lock. For developers the lock means that access to VTScada working copy files, both reading and writing, should not be done without having the lock. The lock is across all applications and system layer VTScada code. The lock prevents two different piece of code from changing the same code such that one piece of code sees inconsistent data while the other code is in the middle of changing it.

This subroutine is a member of the System Library, and must therefore be prefaced by \System\, as shown in the "Format" section. If you are developing a script application, use "System\..." rather than "\System\..." in the function call.

The subroutine returns true (1) if the write was successful and invalid otherwise. If the Settings file or the section does not exist, they will be created. If the VarList parameter is invalid, a blank section will be created. This is a destructive write. That is to say, the entire section of the file is overwritten by the new section, regardless of the contents of either.

Searches performed by this function are case insensitive. Alignment of equal signs in the file is preserved.

Example:

```
If 1 Main;
[
  Vals[0][0] = "OrderlyShutdown";
  Vals[1][0] = 1;
  Vals[0][1] = "ShdownOnLowBattery";
  Vals[1][1] = 1;
  System\WriteSectINI("C:\VTScada\Setup.INI" { Name of file },
    "System" { Name of section },
  Vals { Arrays of data },
  0 { Write file format });
]
```

This causes the variables OrderlyShutdown and ShutdownOnLowBattery in the System section of the Setup.INI file to have their values set to 1 and 1.

X Functions

The sections that follow identify all VTScada functions beginning with "X".

XLoc

Description:	Returns the X window coordinate of the locator (mouse).
Returns:	Numeric
Usage:	Script or steady state.
Function Groups:	Graphics, Locator, Window
Related to:	SetXLoc | SetYLoc | WinXLoc | WinYLoc | YLoc
Format:	XLoc()
Parameters:	None

Example:

```
ZBox(10, YLoc(), XLoc(), 10, 11);
```

This statement will draw a box in the window whose upper left corner is fixed at (10, 10), but whose lower right corner follows the mouse.

XMLAddSchema

Description:	Adds a schema to an XML Processor.
Returns:	Numeric
Usage:	Script Only.
Function Groups:	XML
Related to:	XMLParse | XMLProcessor | XMLWrite | XMLCloneNode | XMLCreateNode | XMLDeleteMember | XMLGetNode | GetXMLNodeArray
Format:	XMLAddSchema(XMLProcessorHandle, NamespaceURI, URL[, ErrorMessageOut])

Parameters:

XMLProcessorHandle

Required. A valid processor, as returned by the function XMLProcessor.

NamespaceURI

Required. A URI that specifies the namespace that the schema represents.

URL

Required. The URL to fetch the schema from.

ErrorMessageOut

An optional text parameter that allows an error message to be returned from the function.

Comments: Each schema will be validated for standards conformance before being added to the cache. Returns 0 if it succeeds, otherwise it returns a numeric specifying a Windows error code and sets the variable named in the optional ErrorMessageOut parameter to a text error message. If the schema load succeeds, the types identified by the schema are added to the factory.

XMLCloneNode

Description: Clones an existing XMLNode, optionally adding additional members.

Returns: The cloned XMLNode, with any additional members supplied.

Usage: Script Only.

Function Groups: XML

Related to: XMLParse | XMLProcessor | XMLAddSchema | XMLWrite | XMLCreateNode | XMLDeleteMember | XMLGetNode | GetXMLNodeArray

Format: XMLCloneNode(XMLNode [, MembersDictionary])

Parameters:

XMLNode

Required. A valid XMLNode.

MembersDictionary

An optional dictionary containing additional members to be added to the cloned node.

Comments: If the optional member dictionary parameter is supplied, any additional members will be added to the cloned node in the same order that they were added to the dictionary.

Example:

```
MembersDict = Dictionary(0);
MembersDict["ISBN"] = XMLCreateNode("01234567890");
MyNode = XMLCloneNode(XMLNode\catalog\book, MembersDict);
```

XMLCreateNode

Description: Creates a new XMLNode.

Returns: An XMLNode.

Usage: Script Only.

Function Groups: XML

Related to: XMLParse | XMLProcessor | XMLAddSchema | XMLWrite | XMLCloneNode | XMLDeleteMember | XMLGetNode | GetXMLNodeArray

Format: XMLNode([Contents, AttributesDictionary, Namespace, MembersDictionary])

Parameters:

Contents

A text value to be placed in the #content member of the XMLNode.

AttributesDictionary

A dictionary containing attributes for the XMLNode

Namespace

A text value containing the namespace for the XMLNode.

MembersDictionary

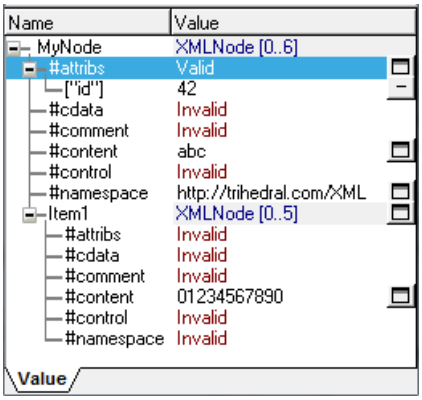
A dictionary containing additional members to be added to the XMLnode.

Comments: If the optional member dictionary parameter is supplied, any additional members will be added to the XMLNode in the same order that they were added to the dictionary. If the #comment or #cdata members require values, they can be assigned after the node has been created.

Example:

```
AttribsDict = Dictionary(0);
AttribsDict["id"] = 42;
MembersDict = Dictionary(0);
MembersDict["Item1"] = XMLCreateNode("01234567890");
MyNode = XMLCreateNode("abc", AttribsDict,
                       http://trihedral.com/XML", MembersDict);
```

Creates the following XMLNode:



XMLDeleteMember

Description: Deletes a member from an XMLNode in-place.

Returns: Nothing.

Usage: Script Only.

Function Groups: XML

Related to: XMLParse | XMLProcessor | XMLAddSchema | XMLWrite | XMLCloneNode | XMLCreateNode | XMLGetNode | GetXMLNodeArray

Format: XMLDeleteMember(XMLNode, MemberName)

Parameters:

XMLNode

Required. The XMLNode to delete the member from.

MemberName

Required. The name of the member to delete.

Comments: The deletion of the member is done "in-place".

XMLGetNode

Description: Returns an XMLNode from a tree.

Returns: The XMLNode specified.

Usage: Script Only.

Function Groups: XML

Related to: XMLParse | XMLProcessor | XMLAddSchema | XMLWrite | XMLCloneNode | XMLCreateNode | XMLDeleteMember | GetXMLNodeArray

Format: XMLGetNode(XMLNode)

Parameters:

XMLNode

Required. The XMLNode to return.

Comments: When isolating a particular XMLNode, to pass to a sub-routine for instance, the automatic subscripting of an XMLNode to provide the value in its #content member means that this is passed to the subroutine.
Thus,

```
GoodXML = Valid(XMLGetNode(XMLData));
```

Is not equivalent to

```
XMLObject = XMLGetNode(XMLData);
GoodXML  = Valid(XMLObject);
```

To defeat the automatic subscripting, use this function.

Example:

Given an XMLNode named "MyNode" as created with the following code:

```
AttribsDict = Dictionary(0);
AttribsDict["id"] = 42;
MembersDict = Dictionary(0);
MembersDict["Item1"] = XMLCreateNode("01234567890");
MyNode = XMLCreateNode("abc", AttribsDict,
                       "http://trihedral.com/XML", MembersDict);
```

Then calling a subroutine as follows will pass the value of the #content member to the subroutine, in this case: "abc":

```
MySub(MyNode);
```

To pass the actual node use the following construct:

```
MySub(XMLGetNode(MyNode));
```

Note that a similar construct uses the address-of operator (&) but then the subroutine will have to de-reference the parameter (using the * operator) on every use.

XMLParse

Description:	Parses the supplied XML using the specified XML Processor.
Returns:	Numeric error code
Usage:	Script Only.
Function Groups:	XML
Related to:	XMLProcessor | XMLAddSchema | XMLWrite | XMLCloneNode | XMLCreateNode | XMLDeleteMember | XMLGetNode | GetXMLNodeArray

Format: XMLParse(XMLProcessorHandle, XMLin [, ErrorMessageOut, XMLNodeTreeOut, NamespaceDictionary])

Parameters:

XMLProcessorHandle

Required. A valid processor, as returned by the XMLProcessor function.

XMLin

Required. The XML text to be parsed. May be either text or a valid stream. If the text supplied (either in the text value or the stream) is identifiable as a URL, the XML is fetched from that URL.

ErrorMessageOut

A text parameter into which a return error message may be placed.

XMLNodeTreeOut

Must be a variable into which the XMLNode tree, created by parsing the XML successfully, will be placed.

NamespaceDictionary

(Return value) A dictionary of namespaces and prefixes found in the parsing of the XML.

Comments: The XML may be either a stream or a text value. Returns 0 if it succeeds. Otherwise returns a numeric specifying a Windows error code and sets the variable named in the optional ErrorMessageOut parameter to a text error message.

Note: In versions of VTS prior to release 10, it is necessary to explicitly slay the XMLNodeTree (the XMLNodeTreeOut parameter to XMLParse) when you are finished with it

The output parameter, XMLNodeTreeOut, will receive a valid XMLNode tree from a validating pro-

cessor even if validation fails, so long as the XML can be parsed. A valid "XMLNodeTreeOut" with a valid "ErrorMessageOut" indicates a parse-able message that fails validation.
A helper function, GetXMLNodeArray, can be used on the result to extract an array of XMLnodes matching a given type name.

XMLProcessor

Description: Creates a new XML Processor.

Returns: XML Handle

Usage: Script Only.

Function Groups: XML

Related to: XMLParse | XMLAddSchema | XMLWrite | XMLCloneNode | XMLCreateNode | XMLDeleteMember | XMLGetNode | GetXMLNodeArray

Format: XMLProcessor(SchemaCacheDictionary)

Parameters:

SchemaCacheDictionary

Required. Must be a variable to hold the Schema Cache Dictionary. All the data types specified by the schemas in the cache will be added to this cache. Provide "Invalid" for a non-validating processor.

Comments: If SchemaCacheDictionary is a valid variable, the XML Processor will be a validating processor, otherwise it will be a non-validating processor. To destroy an XML Processor, invalidate the last reference to it.

XMLWrite

Description: Converts the instance of a type, as specified by XMLNodeTreeIn, into XML.

Returns: Numeric

Usage: Script Only.

Function Groups: File I/O, XML

Related to: XMLParse | XMLProcessor | XMLAddSchema | XMLCloneNode | XMLCreateNode | XMLDeleteMember | XMLGetNode | GetXMLNodeArray

Format: XMLWrite(XMLProcessorHandle, XMLNodeTreeIn, XMLOut, ErrorMessageOut[, NamespaceDictionary, XSLTString])

Parameters:

XLProcessorHandle

Required. A valid processor, as returned by the function XMLProcessor.

XMLNodeTreeIn

Required. An XMLNodeTree such as would be created by XMLParse (XMLNodeTreeOut parameter).

XMLOut

Required. The name of a stream where the XML text will be placed.

ErrorMessageOut

Required. A text parameter into which any return error message is placed.

NamespaceDictionary

An optional dictionary of namespaces and prefixes such as would be created by XMLParse (NamespaceDictionary parameter).

XSLTString

An optional string containing an XSLT (extensible style sheet language transform) to be applied to the output XML.

Comments: The XML is in the form of a stream and is inserted to the

stream passed in XMLOut, starting at its current position. If XMLOut does not contain a stream, a new BufferStream is created and returned in XMLOut. Returns 0 if it succeeds. Otherwise returns a numeric specifying a Windows error code and sets the variable named in the optional ErrorMessageOut parameter to a text error message.
If XSLTString is not a valid XSLT transform, the XML out will not be transformed, but ErrorMessageOut and the return value will be set to indicate the transform failure.

XOr

Description: Returns the bitwise exclusive OR of its parameters.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Bitwise Operation

Related to: And | Not | Or

Format: XOr(Parm1, Parm2)

Parameters:

Parm1, Parm2

Required. Any numeric expressions. If floating point values are supplied, they will be truncated to integers.

Comments: This functions returns the 32 bit bitwise exclusive OR (XOR) of its arguments.

Example:

```
var1 = XOr(0b1100, 0b1010);
```

The value of var1 will be 0b0110.

Y Functions

The sections that follow identify all VTScada functions beginning with "Y".

Year

Description: Returns the year for a given date number.

Returns: Numeric

Usage: Script or steady state.

Function Groups: Time and Date

Related to: Date | DateNum | Day | Month | Today

Format: Year(Date)

Parameters:

Date

Required. Any numeric expression giving the number of days since January 1, 1970.

Comments: This function works in conjunction with the Day and Month functions to decompose a date number into the corresponding day, month and year.

Example:

```
whatYear = Year(8394 { 25 December 1992 });
```

The value of whatYear will be 1992.

YLoc

Description: Returns the Y window coordinate of the locator (mouse).

Returns: Numeric

Usage: Script or steady state.

Function Groups: Graphics, Locator, Window

Related to: SetXLoc | SetYLoc | WinXLoc | WinYLoc | XLoc

Format: YLoc()

Parameters: None

Example:

```
If Change(YLoc(), 10);
[
  ...
]
```

The action trigger will become true when the Y-coordinate of the mouse changes by more than 10 pixels, and the script will then execute.

Z Functions

The sections that follow identify all VTScada functions beginning with "Z".

ZBar

Description: Draws a layered bar in a window.

Returns: Nothing

Usage: Steady State only.

Function Groups: Graphics

Related to: Bar | Box | GUIRectangle | ZBox

Format: ZBar(Left, Bottom, Right, Top, Color)

Parameters:

Left

Required. Any numeric expression for the left side coordinate of the bar.

Bottom

Required. Any numeric expression for the bottom side

coordinate of the bar.

Right

Required. Any numeric expression for the right side coordinate of the bar.

Top

Required. Any numeric expression for the top side coordinate of the bar.

Color

Required. May be any of the following:

- An RGB string in the form, "<RRGGBB>"
- A Constants for System Colors
- A VTScada Color Palette number from the palette.
- -1 (transparent)

Comments: Although this is a layered graphic, it cannot be edited using the Idea Studio. It is for use within text mode editing only.

Example:

```
ZBox(10, 200, 210, 10 { Bounding box for box },
     0 { Box is black });
ZBar(10, 200, 210,
     Limit(Scale(fluid, 0 { min }, 2000 { max },
     200 { scaled min }, 10 { scaled max }),
     10 { Min value from Scale },
     200 { Max value from Scale }),
     { Bounding box for bar }
     10 { Color of bar });
```

These two statements simulate a tank that is outlined in white (using ZBox) and whose changing fluid level is represented in the ZBar statement by scaling and limiting the value of fluid. Notice that the Scale function converts (and inverts) the value from a real world fluid level to window coordinates. Because its return value is not limited by any of its parameters, the Limit function is used to declare absolute limits. Even if

the value of fluid goes beyond its limits, the bar will not be drawn outside of the box.

ZBox

Description: Draws a layered box in a window.

Returns: Nothing

Usage: Steady State only.

Function Groups: Graphics

Related to: Bar | Box | GUIRectangle | ZBar

Format: ZBox(Left, Bottom, Right, Top, Color)

Parameters:

Left

Required. Any numeric expression for the left side coordinate of the box.

Bottom

Required. Any numeric expression for the bottom side coordinate of the box.

Right

Required. Any numeric expression for the right side coordinate of the box.

Top

Required. Any numeric expression for the top side coordinate of the box.

Color

Required. May be any of the following:

- An RGB string in the form, "<RRGGBB>"
- A Constants for System Colors
- A VTScada Color Palette number from the palette.
- -1 (transparent)

Comments: Although this is a layered graphic, it cannot be edited using the Idea Studio. It is for use within text mode editing only.

Example:

```
ZBox(10, 110, 220, 10 { Bounding box for box },
     12 { Box is red });
```

This statement draws a (hollow) box in the window.

ZButton

Description: Draws a layered button in a window and returns true when selected.

Returns: Boolean

Usage: Steady State only.

Function Groups: Graphics

Related to: GUIButton

Format: ZButton(Left, Bottom, Right, Top, Label, FocusID [, Font])

Parameters:

Left

Required. Any numeric expression for the left side coordinate of the button.

Bottom

Required. Any numeric expression for the bottom side coordinate of the button.

Right

Required. Any numeric expression for the right side coordinate of the button.

Top

Required. Any numeric expression for the top side coordinate of the button.

Label

Required. Any text expression for the button label.

FocusID

Required. Any numeric expression for the focus ID number. This is used to force the focus to this item. If this value is 0, the button text will be grayed, the button cannot be focused and will not respond to mouse clicks. Values above or below zero allow the control to be used. This value is stored in a short. If greater than 32767, the button will not be visible.

Font

An optional parameter that is any expression for the font value to use. If this value is omitted, the default value of system font is used.

Comments:

Although this is a layered graphic, it cannot be edited using the Idea Studio. It is for use within text mode editing only. The colors for this graphic are taken from Microsoft Windows™.

This function behaves like a Pick with the pressing of the left mouse button being the button combination to be detected. It will return 1 on the pressing of the left mouse button.

Example:

```
If ZButton(10, 70, 110, 40 { Bounding box for button },
           "OK" { Text label on button },
           1 { Focus ID })
           DoneState;
[
  ...
]
```

This action trigger will become true when the left mouse button clicks on the button drawn in the window. At that point the script will execute and then there will be a state change to DoneState.

ZColorChange

Description: Changes one color within a region to another color. This is an older function, intended for use on objects that used a color range, such as pipes prior to release 11 of VTScada.

Returns: Nothing

Usage: Steady State only.

Function Groups: Color, Graphics

Related to: GetSystemColor | PixelColor

Format: ZColorChange(Left, Bottom, Right, Top, Original, New)

Parameters:

Left

Required. Any numeric expression for the left side coordinate of the change region.

Bottom

Required. Any numeric expression for the bottom side coordinate of the change region.

Right

Required. Any numeric expression for the right side coordinate of the change region.

Top

Required. Any numeric expression for the top side coordinate of the change region.

Original

Required. Any one of the colors of the existing region.

New

Required. Any one of the colors which will replace the old.

Comments: Although this is a layered graphic, it cannot be edited using the Idea Studio. It is for use within text mode editing

only. A solid pattern should be used if you wish to change the color back to its original color.

The order of graphics statements are an important factor in determining which objects will have their color changed. The ZColorChange function affects all objects underneath it, which means, any object whose graphic statement is before the ZColorChange statement (including images that are displayed using a Load statement). Any object whose statement follows the ZColorChange statement will be unaffected. The background color will also be affected by ZColorChange.

Example:

```
ZBar(100, 500, 500, 40, 12 { Draws a large red bar });
ZColorChange(10, 210, 210, 10 { Bounding box for change },
             12, 14 { Change all red in to yellow });
ZBar(20, 60, 60, 20, 12 { Draws a small red bar });
```

The ZColorChange statement here will cause the upper left corner of the first large bar to be yellow, while the rest of the bar (outside the bounding box for ZColorChange) will remain red. The second bar, although drawn inside of the ZColorChange's bounding box, will be drawn in red, not yellow, because its statement follows the statement that effects the color change.

ZEditField

Note: Deprecated. Do not use in new code.

Description: Draws a layered text edit field in a window and returns a status value. Not editable when viewed on a VTScada Internet Client.

Returns: Numeric

Usage: Steady State only.

Function Groups: Editor, Graphics

Related to: Cast | WinEditCtrl

Format: ZEditField(Left, Bottom, Right, Top, TextVar, Length, Font, FocusID)

Parameters:

Left

Required. Any numeric expression for the left side coordinate of the field.

Bottom

Required. Any numeric expression for the bottom side coordinate of the field.

Right

Required. Any numeric expression for the right side coordinate of the field.

Top

Required. Any numeric expression for the top side coordinate of the field.

TextVar

Required. Must be a variable. The result of the text editing will be stored here. The value in this field will be automatically inserted into the field.

Length

Required. Any numeric expression for the number of characters allowed in this field.

Font

Required. Any expression for the font value to use.

FocusID

Required. Any numeric expression for the focus ID number. This is used to force the focus to this item. Values below zero will render this control invisible. A value of zero makes this control visible, but not editable. With values above zero,

the control will be editable.

Comments: Although this is a layered graphic, it cannot be edited using the Idea Studio. It is for use within text mode editing only. The colors for this graphic are taken from Microsoft Windows™.

The return values for this function are as follows

Return Value	Meaning
0	Internal buffer changed
1	<CR> pressed
2	Focus has been lost

Although data may be entered in the field, TextVar is not set until <RETURN> is pressed, or a change in focus occurs. This is crucial when setting other variables based on TextVar. As their values may not be set prior to a state change if <RETURN> has not been pressed.

This function does not check the input type, but converts all data to text. Data that is required to be of a type other than text must be converted to that type using the Cast function.

Note: Within an Anywhere Client session, this function does nothing.

Note: if the application is to be used on a VTScada Internet Client, WinEditCtrl should be used instead of ZEditField.

Example:

```
ZEditField(100, 500, 200, 470 { Bounding box for field },
           inputVal { Input value },
           3 { Max chars allowed in input },
           0, 1 { Default font and focus ID });
```

This statement will accept up to 3 characters of input and store it in inputVal. If inputVal has a default value, that value will be shown in the white edit field box until it is changed.

ZGrid

Description: Draws a layered point grid in a window.

Returns: Nothing

Usage: Steady State only.

Function Groups: Graphics

Related to: Grid

Format: ZGrid(Left, Bottom, Right, Top, Color, XSpace, YSpace)

Parameters:

Left

Required. Any numeric expression for the left side coordinate of the grid.

Bottom

Required. Any numeric expression for the bottom side coordinate of the grid.

Right

Required. Any numeric expression for the right side coordinate of the grid.

Top

Required. Any numeric expression for the top side coordinate of the grid.

Color

Required. May be any of the following:

- An RGB string in the form, "<RRGGBB>"
- A Constants for System Colors
- A VTScada Color Palette number from the palette.
- -1 (transparent)

XSpace

Required. Any numeric expression giving a positive integer value for the spacing of the x-grid.

YSpace

Required. Any numeric expression giving a positive integer value for the spacing of the y-grid.

Comments: Although this is a layered graphic, it cannot be edited using the Idea Studio. It is for use within text mode editing only.

Example:

```
ZGrid(100, 200, 200, 100 { Bounding box for grid },
      12 { Grid is light red },
      5, 5 { X and Y spacing for grid });
```

This places a light red 5x5 point-grid on the screen in the area bounded by the first four parameters.

ZLine

Description: Draws a line between given x and y coordinates in a win-dow.

Returns: Nothing

Usage: Steady State only.

Function Groups: Graphics

Related to: GUIPolygon | Line

Format: ZLine(X1, Y1, X2, Y2, Pen)

Parameters:

X1, Y1

Required. Any numeric expressions for the X and Y coordinates of the first endpoint of the line.

X2, Y2

Required. Any numeric expressions for the X and Y coordinates of the second endpoint of the line.

Pen

Required. Any expression that returns the PEN object

to use to draw the line.

A color value is also acceptable here, which may be any of the following:

- An RGB string in the form, "<RRGGBB>"
- A Constants for System Colors
- A VTScada Color Palette number from the palette.
- -1 (transparent)

Comments: Although this is a layered graphic, it cannot be edited using the Idea Studio. It is for use within text mode editing only.

Example:

```
ZLine(0, 0 { First endpoint for line },
      799, 599 { Second endpoint for line },
      14 { Line is yellow });
```

This statement draws a yellow line diagonally from the upper left corner to the lower right corner of the window.

ZPipe

Note: Deprecated. Do not use in new code.

Description: Draws a layered three-dimensional pipe in a window.

Returns: Nothing

Usage: Steady State only.

Function Groups: Graphics

Related to: GUIPipe | Pipe

Format: ZPipe(XArrayElem, YArrayElem, N, LowIndex, HighIndex, PixelWidth)

Parameters:

XArrayElem

Required. Any numeric expression that specifies the

starting array element. The array contains the X coordinates of the path along which to draw the pipe. If processing a multidimensional array, the usual rules apply to decide which dimension should be examined.

YArrayElem

Required. Any numeric expression that specifies the starting array element. The array contains the Y coordinates of the path along which to draw the pipe. If processing a multidimensional array, the usual rules apply to decide which dimension should be examined.

N

Required. Any numeric expression giving the number of array elements to use to define the path of the pipe.

LowIndex

Required. Any numeric expression specifying the low index into the current color palette. It is used in conjunction with the next parameter to adjust the brightness and contrast of the shading.

HighIndex

Required. Any numeric expression specifying the high index into the current color palette. It is used in conjunction with the previous parameter to adjust the brightness and contrast of the shading.

PixelWidth

Required. Any numeric expression specifying the width of the shaded pipe in pixels and is subject to applicable scaling factors.

Comments:

Although this is a layered graphic, it cannot be edited using the Idea Studio. It is for use within text mode editing only.

The pipe will be drawn with a miter effect, such that pipe segments meet at 45-degree angles.

Example:

```
ZPipe(x[0], y[0] { First endpoint for pipe },
      8 { Number of vertices },
      176, 239 { Very dark gray to very light gray },
      24 { Width of pipe in pixels });
```

ZText

Description: Draws a layered text label in a window.

Returns: Nothing

Usage: Steady State only.

Function Groups: Graphics

Related to: GUIText | Format

Format: ZText(Left, Bottom, Value, Color, Font)

Parameters:

Left

Required. Any numeric expression for the left side coordinate of the text.

Bottom

Required. Any numeric expression for the bottom side coordinate of the text.

Value

Required. Any text expression to display on the screen.

Color

Required. May be any of the following:

- An RGB string in the form, "<RRGGBB>"
- A Constants for System Colors
- A VTScada Color Palette number from the palette.
- -1 (transparent)

Font

Required. Any expression for the font value to use. A numeric zero (0) may be used to get the default system font.

Comments: Although this is a layered graphic, it cannot be edited using the Idea Studio. It is for use within text mode editing only.

Examples:

```
ZText(100, 100 { Lower left corner of text },
      "Hello World" { Text to display },
      2 { Text is green},
      0 { Use default font });
```

This places the string "Hello World" on the screen at (100, 100) with white text in the system font.

```
ZText(500, 100 { X, Y coordinates },
       Format(0, 2, DeadBand(pressure, 5)) { Displayed value },
       0, 0);
```

This displays the value of variable, "pressure" on the screen. However, the display is only updated when pressure changes by more than 5 (higher or lower) from the last update.

Index

C

D

E

F

G

H

I

K

N

O

P

Q

R

S

T

U

V

W

X

Y

Z