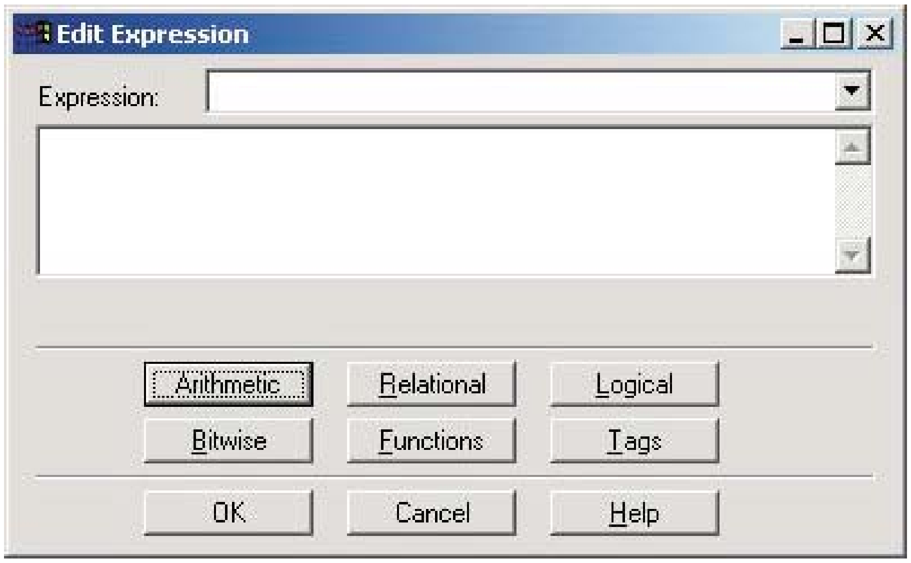
USER'S MANUAL

Expression Editor - smar First in Fieldbus

Expression Editor



MAY / 06

Expression Editor

VERSION 8

FOUNDATION

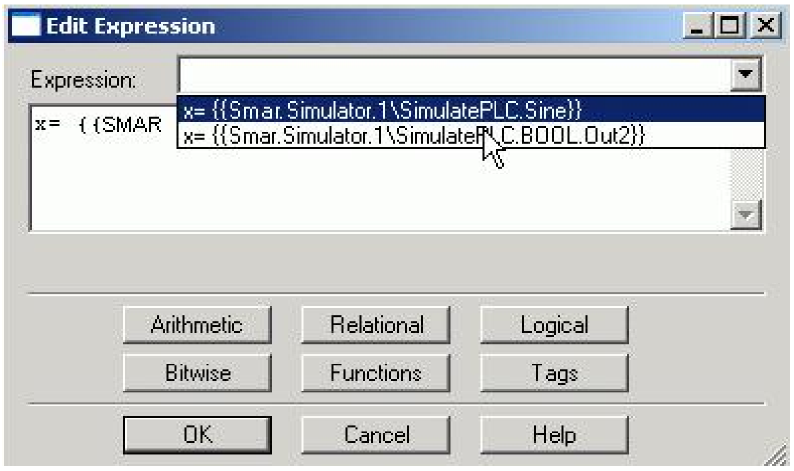
P V I E W E X E M E

TABLE OF CONTENTS

CREATING EXPRESSIONS USING THE EXPRESSION EDITOR ***5***
WRITING EXPRESSIONS 5
STRINGS IN EXPRESSIONS 5
STRINGS COMPARISON 6
DATA TYPE CONVERSION 6
POINT EXTENSION SYNTAX 6
ARITHMETIC 7
LOGICAL 8
BITWISE 9
FUNCTIONS 12
CONDITIONAL 13
MATHEMATICAL 14
STRING 15
CONSTANT 19
OPC QUALITY 21
TAGS 22
SELECTING ALARM ATTRIBUTES 26

CREATING EXPRESSIONS USING THE EXPRESSION EDITOR

The **Expression Editor,** shown below, is available to assist you in creating expressions for your ProcessView applications. The window is resizable and can be stretched as well as maximized or minimized. The drop-down list at the top of the **Edit Expression** dialog box keeps track of the last 50 expressions you have entered. The expression entered most recently is the first one in the drop-down list.



Expression Editor

Writing Expressions

An **expression** is a string that defines and evaluates a data connection between a client and an OPC server. During runtime mode, OPC servers resolve the data value for the expression. To indicate that a data connection is an expression, precede the string with the "**x=**" token, as shown below:

x={{Smar.Simulator.1\SimulatePLC.PumpSpeed}}
You can either type your expressions directly into the text box of the **Edit Expression** dialog box, or you can use the symbols and functions provided that help you use the proper string syntax when writing expressions. The following categories are available:

- Arithmetic
- Relational
- Logical
- Bitwise
- Functions
- Tags

Strings in Expressions

Strings can be used in expressions. Constant strings are delimited with double quotation marks. For example:
“Hello World”

Constant strings can also be enclosed between $” and “$ characters. For example:
$”Hello World”$

Strings Comparison

When comparing strings or numeric data coming from the server as strings, the comparison is based on the orders of the characters. For example, the expression:

x="world" > "hello"

evaluates as true, because the character "w" comes after the character "h" in alphabetical order.

Sometimes the string comparison may be misleading. For example:

x="20" > "100"

evaluates as true, because the character "2" comes after the character "1" in the character table. Of course, if there were an expectation of a numeric comparison, 20 < 100 and the above expressions might seem to be evaluated incorrectly, but they are not.

Data Type Conversion

Expressions allow calculations to be performed on incoming data. An OPC server can provide data in one or more data types, such as "float," "long," "integer," "string," etc.

Some OPC servers provide numeric data as a string. For example, the numeric value 20 may be provided as the string "20" (character "2" followed by character "0"). This may be lead an incorrect expression evaluation (see the "String Comparison" section above).

There is a workaround: If you add a numerical zero to each of the tags, the logic operators will work properly. For example:

x=({{JC.N1OPC.1.0\HDQTRS\sys2\ad-3.Present Value}}+0) >
({{JC.N1OPC.1.0\HDQTRS\sys2\ad-4.Present Value}}+0)
An alternative way is to change the OPC server so that it sends the strings with a fixed number of digits with leading zeros, or to use DataWorX registers for a conversion from a string to a number.

Some of the functions provided in the Expression Editor use parameters of type numeric. When possible, the Expression Editor automatically converts the string into a number. For example:
The string "20" can automatically be converted to the number 20.
If the string contains alphabetic characters or symbols, then the automatic conversion is impossible.

For example:
The string "20hello" cannot be converted into a number.

Even if the string contains only numeric characters and valid symbols, there may still be cases in which an automatic conversion is not possible. For example:

The string "123.45.23" cannot be converted into a number because it contains two decimal separators. Sometimes strings and numbers are mixed in expressions. In this case, the Expression Editor attempts to convert the string into a number. For example:
Str + Number, where Str is a string convertible to a Number, results in a Number.
If the string is not convertible into a number, then the result will have a bad quality.

The following is an example of a valid expression:
x=5+"6"

Point Extension Syntax

The **Point Extension Syntax (PES)** allows for retrieving additional information related to OPC tags, such as quality and timestamp.

To use the PES:

1. Prefix your tag name with "tag:"
2. Postfix your tag name with "#" followed by a PES token.

Valid PES tokens are:

- **quality (returns the OPC quality associated with the tag value)**
- **timestamp** (returns the timestamp associated with the tag value)

The following are example expressions using a valid PES request:

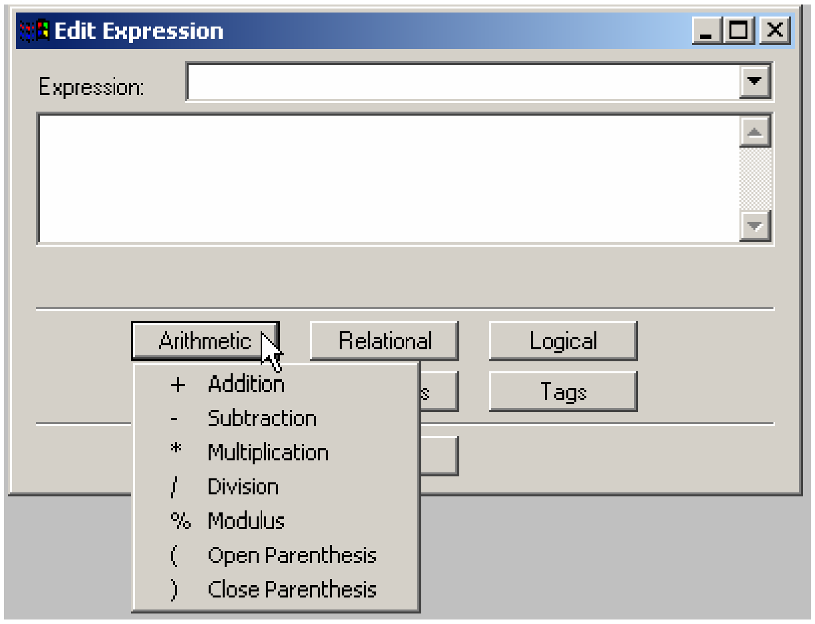
- **tag:Smar.Simulator\SimulatePLC.Ramp#timestamp**
- **tag:Smar.Simulator\SimulatePLC.Ramp#quality**
- **tag:\\pc1\Smar.Simulator\SimulatePLC.Ramp#timestamp**
- **tag:\\pc1\Smar.Simulator\SimulatePLC.Ramp#quality**

Please see the “OPC Quality” section below for additional information about OPC quality.

NOTE
Sometimes it may be necessary to enforce the “request data type” to a specific type, such as “string,” in order to display the extended syntax information in a process point

Arithmetic

The **Arithmetic** menu symbols are shown in the figure below.



Arithmetic Symbols

The symbols '+', '-', '*', '/' and '%' use the following format:
parameter **symbol** parameter

Where

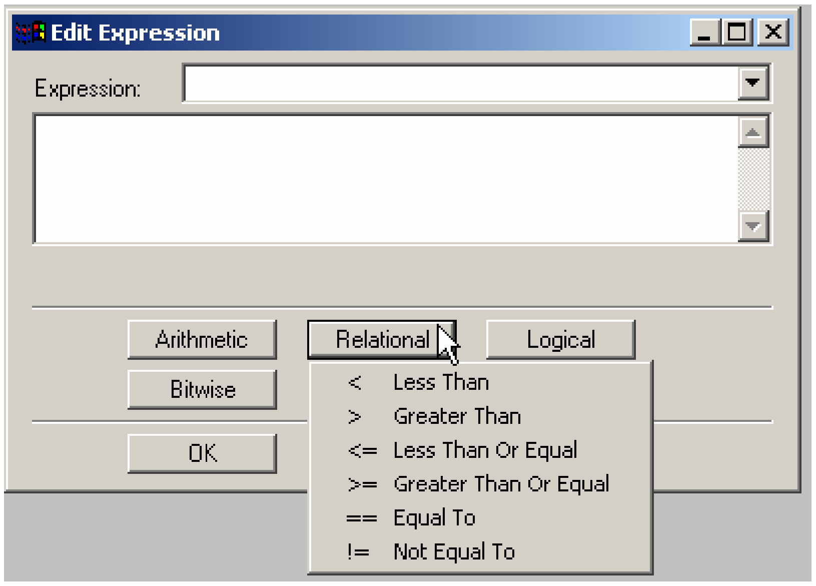
PARAMETER	A local variable, an OPC tag, a constant, or another expression
SYMBOL	+ or - or * or / or %

Result
The expression results in a number of any type (float, long, etc.).

Examples

SYMBOL	DESCRIPTION	EXAMPLE	RESULT
+	Addition	~~var1~~ + ~~var2~~	8+3 = 11
-	Subtraction	~~var1~~ - ~~var2~~	8-3 = 5
*	Multiplication	~~var1~~ * ~~var2~~	8*3 = 24
/	Division	~~var1~~ / ~~var2~~	8/3 = 2.66667
%	Calculates the remainder after division	~~var1~~ % ~~var2~~	8%3 = 2
(and)	Gives precedence to parts of the calculation	~~var1~~ / (~~var2~~ + ~~var3~~)	8/(3+2) = 1.6

Relational
The **Relational** menu symbols are shown in the figure below.



Relational Symbols

The symbols '<', '>', '<=', '>=', '==' and '!=' use the following format:
parameter **symbol** parameter

Where

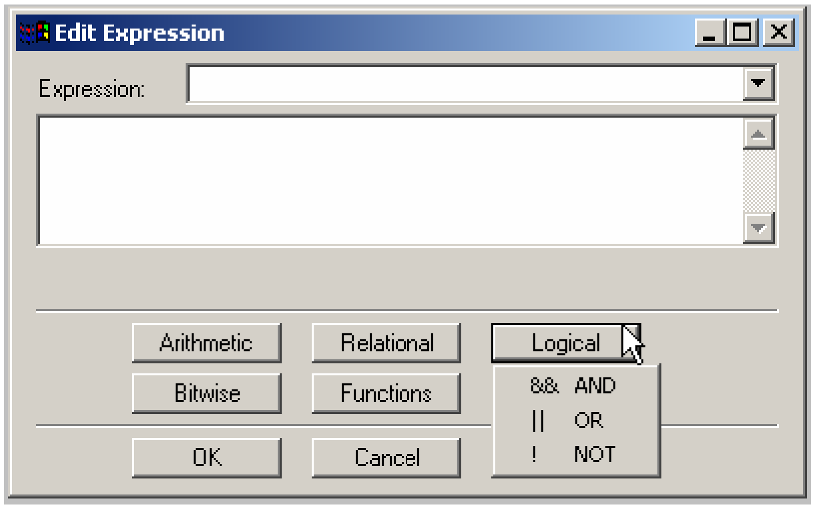
PARAMETER	A local variable, an OPC tag, a constant, or another expression
SYMBOL	< or > or <= or >= or == or !=

Result
The expression results in a Boolean value (0 or 1).

Examples

Symbol	Description	Example	Result
<	Less than	~~var1~~ < ~~var2~~	8<3 = 0
>	Greater than	~~var1~~ > ~~var2~~	8>3 = 1
<=	Less than or equal to	~~var1~~ <= ~~var2~~	8<=3 = 0
>=	Greater than or equal to	~~var1~~ >= ~~var2~~	8>=3 = 1
==	Equal to	~~var1~~ == ~~var2~~	8==3 = 0
!=	Not equal to	~~var1~~ != ~~var2~~	8!=3 = 1

Logical
The **Logical** menu symbols are shown in the figure below.



Logical Symbols

The symbols '&&' and '||' use the following format:
parameter **symbol** parameter

The symbol '!' uses the following format:
symbol parameter

Where

PARAMETER	A local variable, an OPC tag, a constant, or another expression
SYMBOL	&& or || or !

Result
The expression results in a Boolean value (0 or 1).

Truth table

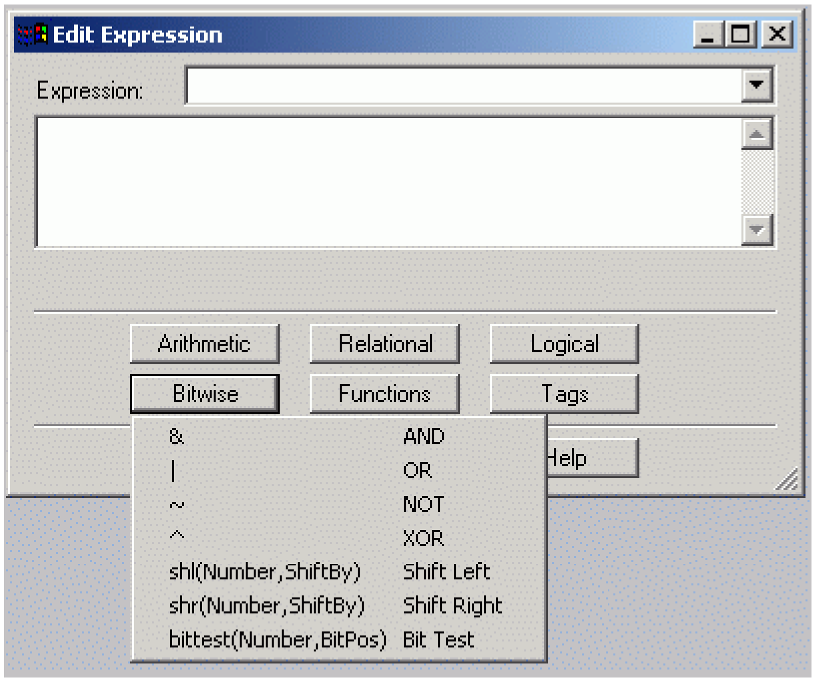
`~~var1~~`	0		not 0	
`~~var2~~`	0	not 0	0	not 0
`~~var1~~ && ~~var2~~`	0	0	0	1
`~~var1~~ || ~~var2~~`	0	1	1	1
`!~~var1~~`	1	1	0	0

Examples

SYMBOL	DESCRIPTION	EXAMPLE	RESULT
&&	And	~~var1~~ && ~~var2~~	8 && 3 = 1
||	Or	~~var1~~ || ~~var2~~	8 || 3 = 1
!	Not	!~~var1~~	!8 = 0

Bitwise

The **Bitwise** menu symbols are shown in the figure below.



&	AND
|	OR
~	NOT
^	XOR
shl(Number,ShiftBy)	Shift Left
shr(Number,ShiftBy)	Shift Right
bittest(Number,BitPos)	Bit Test

Bitwise Symbols

The symbols '&', '|', and '^' of the bitwise group use the following format:
parameter **symbol** parameter

Where

PARAMETER	A local variable, an OPC tag, a constant, or another expression

The symbol '~' of the logical group uses the following format:
~ parameter

Where

PARAMETER	A local variable, an OPC tag, a constant, or another expression

The symbols 'shl' and 'shr' of the bitwise group use the following format:
symbol (number, shift by)

Where

SYMBOL	shl or shr
NUMBER	A local variable, an OPC tag, a constant, or another expression
SHIFT BY	The number of bits to shift

The symbol 'bittest' of the bitwise group uses the following format:
BitTest (number, bit position)

NUMBER	A local variable, an OPC tag, a constant, or another expression
BIT POSITION	The position of the bit to test. A bit position of "0" indicates the "less significant" bit

Examples

SYMBOL	DESCRIPTION	EXAMPLE	RESULT
&	Bit And	~~var1~~ & ~~var2~~	8 & 3 = 0
|	Bit Or	~~var1~~ | ~~var2~~	8 | 3 = 11
^	Bit eXclusive Or	~~var1~~ ^ ~~var2~~	8^3=11
shl	Bit shift left	shl(~~var1~~,3)	8<<3=64
shr	Bit shift right	shr(~~var1~~,3)	8>>3=1
~	Not (two's complement)	~(~~var1~~)	!8 = -9
bittest	Bit Test	bittest (5 , 0)	1

Examples

The following two examples use the variables ~~var1~~ and ~~var2~~ in several expressions that use bitwise symbols.

In Example 1, the decimal values for these variables are:

- ~~var1~~ = 8
- ~~var2~~ = 10

In **Example 2,** the decimal values for these variables are:

- ~~var1~~ = 96
- ~~var2~~ = 8

NOTE
The binary values are also listed in the tables below. The representation chosen is 16 bits per variable.

Variable Values for Example 1 and Example 2

	Example 1 Values (Decimal) Binary	Example 2 Values (Decimal) Binary
~~var1~~	(8) 0000.0000.0000.1000	(96) 0000.0000.0110.0000
~~var2~~	(10) 0000.0000.0000.1010	(8) 0000.0000.0000.1000

Example 1

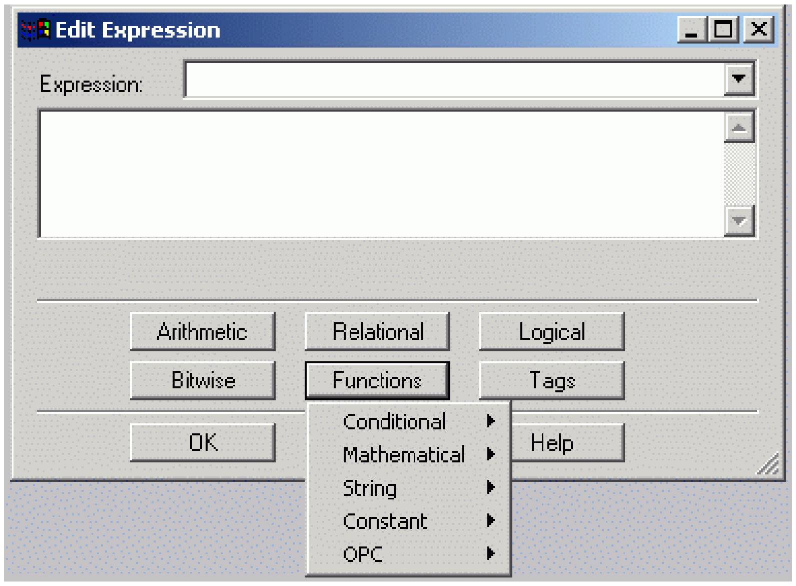
Expression ~~var1~~ = 8 ~~var2~~ = 10	Result (Decimal) Binary
~~var1~~ & ~~var2~~	(8) 0000.0000.0000.1000
~~var1~~ | ~~var2~~	(10) 0000.0000.0000.1010
~~var1~~ ^ ~~var2~~	(2) 0000.0000.0000.0010
shl (~~var1~~,3)	(64) 0000.0000.0100.0000
shr (~~var1~~,3)	(1) 0000.0000.0000.0001
~(~~var1~~)	(-9) 1111.1111.1111.0111
bittest(~~var1~~,3)	(1) 0000.0000.0000.0001

Example 2

Expression ~~var1~~ = 96 ~~var2~~ = 8	Result (Decimal) Binary
~~var1~~ & ~~var2~~	(0) 0000.0000.0000.0000
~~var1~~ | ~~var2~~	(104) 0000.0000.0110.1000
~~var1~~ ^ ~~var2~~	(104) 0000.0000.0110.1000
shl (~~var1~~,3)	(768) 0000.0011.0000.0000
shr (~~var1~~,3)	(12) 0000.0000.0000.1100
~(~~var1~~)	(-97) 1111.1111.1001.1111
bittest(~~var1~~,3)	(0) 0000.0000.0000.0000

Functions

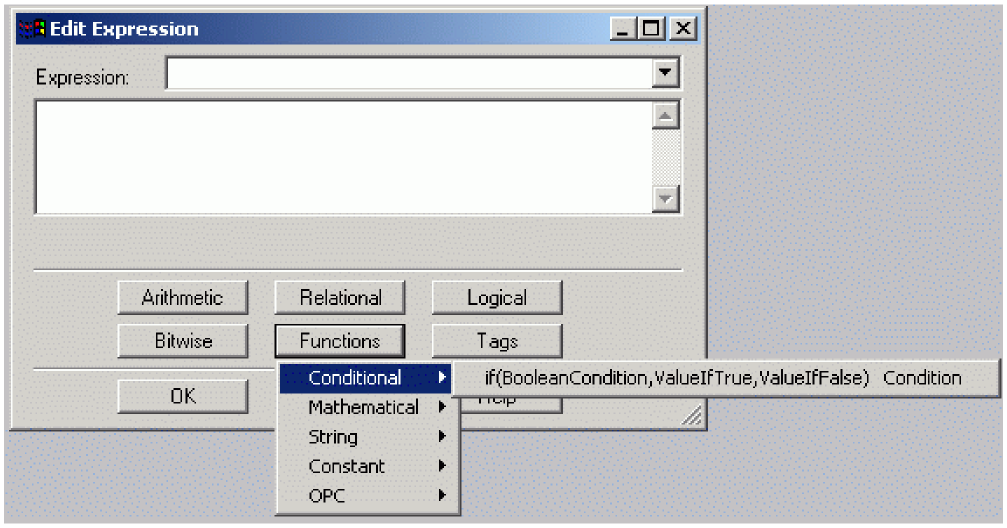
The **Functions** menu options are shown in the figure below.



Functions Menu Options

Conditional

The **Conditional** function is shown in the figure below.



Conditional "If" Function

The symbol 'if' is a conditional statement that uses the following format:
symbol (parameter,parameter,parameter)

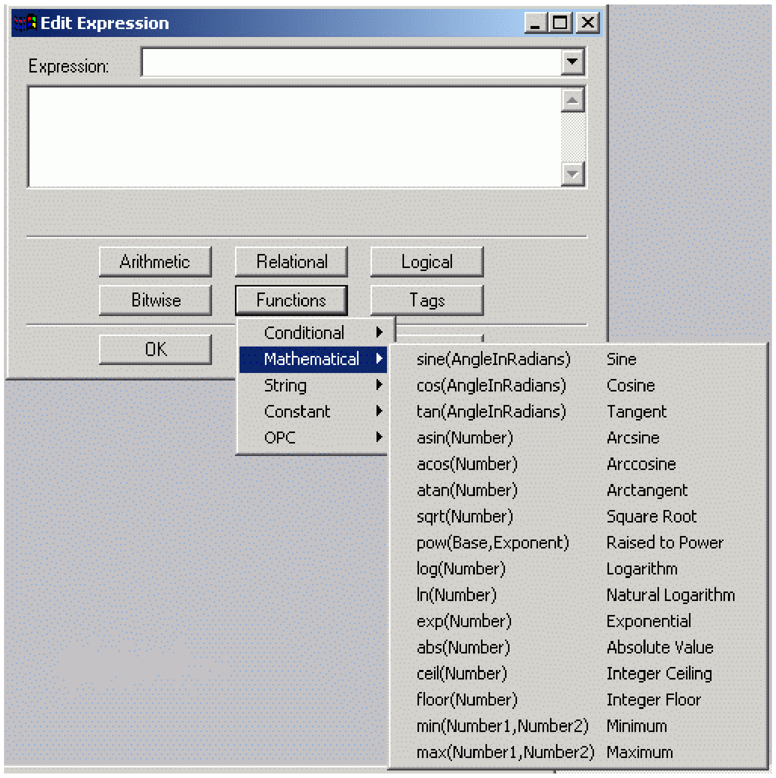
The **if** function is used to evaluate an expression and assign a value based on the result of the evaluation. The syntax of the "if" function is as follows:
result=If (Condition,value1,value2)
If “condition” is TRUE, then the result will be "value1." If “condition” is FALSE then the result will be "value2."
"Condition," "value1" and "value2" can be constant values, variables, process points, or expressions.

Note that the numerical value 0 will be evaluated as FALSE, while all the other values will be evaluated as TRUE. The following table shows some examples of "if" functions.

FUNCTION	COMMENTS
If (~~var~~>0, 1, 0)	If the local variable ~~var~~ is a positive number, then the expression will return 1. In all other cases the expression will return 0.
If (~~var~~<0, 10, 55)	If the local variable ~~var~~ is a negative number, then the expression will return 10. In all other cases the expression will return 55.
If (~~pressure~~>100, ~~level~~, ~~pumpstatus~~)	If the local variable ~~pressure~~ is greater than 100, then the expression will return the value of the variable ~~level~~. In all other cases the expression will return the value of the variable ~~pumpstatus~~.
If (~~pump_on~~==1,0,1)	If the value of ~~pump_on~~ is 1, then the expression will return 0. In all other cases the expression will return 1.
If (~~pump_on~~==1, 1, max (~~var1~~,~~var2~~))	If the value of ~~pump_on~~ is 1, then the expression will return 1. In all other cases the expression will return the maximum between the value of ~~var1~~ and ~~var2~~.
If ({{MYTAG}} >=~~min~~ , {{MYTAG}}, ~~min~~)	If the value of the process point MYTAG is greater than or equal to the value of the local variable ~~min~~, then the expression will return the value of the process point itself. In all other cases the expression will return the value of the local variable ~~min~~.

Mathematical

The **Mathematical** functions are shown in the figure below.


Edit Expression
Expression:
Arithmetic
Relational
Logical
Bitwise
Functions
Tags
OK
Conditional
Mathematical
String
Constant
OPC
sine(AngleInRadians) Sine
cos(AngleInRadians) Cosine
tan(AngleInRadians) Tangent
asin(Number) Arcsine
acos(Number) Arccosine
atan(Number) Arctangent
sqrt(Number) Square Root
pow(Base,Exponent) Raised to Power
log(Number) Logarithm
ln(Number) Natural Logarithm
exp(Number) Exponential
abs(Number) Absolute Value
ceil(Number) Integer Ceiling
floor(Number) Integer Floor
min(Number1,Number2) Minimum
max(Number1,Number2) Maximum

Mathematical Functions

The symbols 'sin', 'asin', 'cos', 'acos', 'tan', 'atan', 'log', 'ln', 'exp', 'sqrt', 'abs', 'ceil', and 'floor' use the following format:
symbol (parameter)

The symbols 'pow', 'min', and 'max' use the following format:
symbol (parameter,parameter)

Note: The parameter should be a valid number or a string convertible to a number (see the "Data Type Conversion" section above).
Note also that the parameter should belong to the domain of values acceptable for the function in which it is used.
For example: function asin(variable) exists only when –1<= variable <= 1.
For additional information, please refer to a mathematical reference book.

Note: Since the constant pi, presented here, actually is the rounded number pi, the expression such as tan(PI*N/2), where N is an odd number, is considered here as a valid expression. Please see the "Constant" section below.

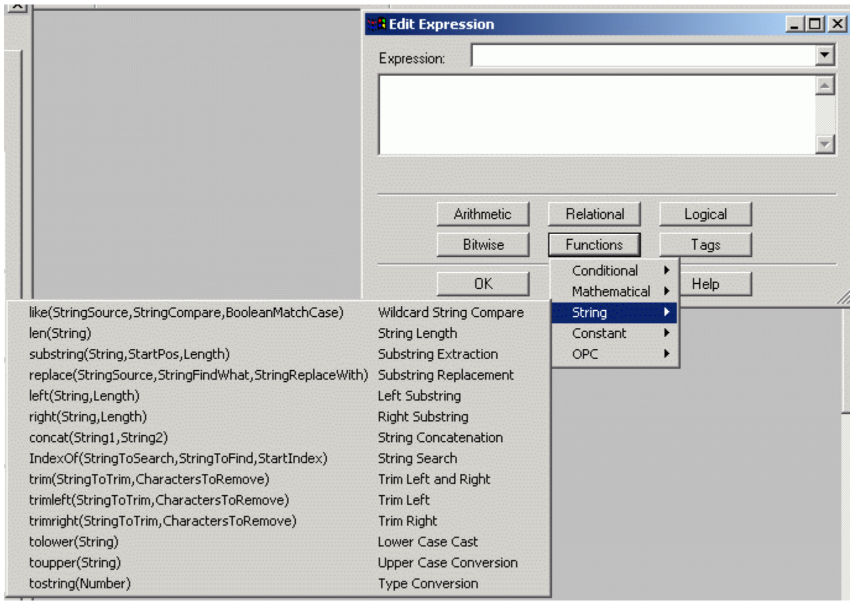
Examples

SYMBOL	DESCRIPTION	EXAMPLE	RESULT
sin	sine of an angle in radians	sin(~~var1~~)	sin(0.785)=0.71
cos	cosine of an angle in radians	cos(~~var1~~)	cos(0.785)=0.71
tan	tangent of an angle in radians	tan(~~var1~~)	tan(0.785)=1.0
asin	arc sine returns an angle in	asin(~~var1~~)	asin(0.5)=0.52

SYMBOL	DESCRIPTION	EXAMPLE	RESULT
	radians		
acos	arc cosine returns an angle in radians	acos(~~var1~~)	acos(0.5)=1.05
atan	arc tangent returns an angle in radians	atan(~~var1~~)	atan(1)=0.785
sqrt	Returns the square root	sqrt(~~var1~~)	sqrt(100)=10
pow	Returns value 1 raised to the power value 2	pow(~~var1~~,~~var2~~)	pow(100,1.5)=1000
log	10 based logarithm	log(~~var1~~)	log(100)=2
ln	*e* based logarithm	ln(~~var1~~)	ln(7.389)=2
exp	Exponential	exp(~~var1~~)	exp(2)=7.389
abs	Absolute value	abs(~~var1~~)	abs(-1)=1
ceil	Integer ceiling	ceil(~~var1~~)	ceil(7.39)=8
floor	Integer floor	floor(~~var1~~)	floor(7.39)=7
min	Lowest value of two	min(~~var1~~,~~var2~~)	min(10,5)=5
max	Highest value of two	max(~~var1~~,~~var2~~)	min(10,5)=10

String

The **String** functions are shown in the figure below.



like(StringSource,StringCompare,BooleanMatchCase)	Wildcard String Compare
len(String)	String Length
substring(String,StartPos,Length)	Substring Extraction
replace(StringSource,StringFindWhat,StringReplaceWith)	Substring Replacement
left(String,Length)	Left Substring
right(String,Length)	Right Substring
concat(String1,String2)	String Concatenation
IndexOf(StringToSearch,StringToFind,StartIndex)	String Search
trim(StringToTrim,CharactersToRemove)	Trim Left and Right
trimleft(StringToTrim,CharactersToRemove)	Trim Left
trimright(StringToTrim,CharactersToRemove)	Trim Right
tolower(String)	Lower Case Cast
toupper(String)	Upper Case Conversion
tostring(Number)	Type Conversion

String Functions

In all the string functions, which are listed in the table below, the position of the first character is zero. Thus, if a position is returned or a starting position is provided as a parameter, then they all refer to a zero indexed string.

STRING FUNCTION	DESCRIPTION
like(StringSource,StringCompare,BooleanMatchCase)	Wildcard string compare (see note below)
len(String)	Returns the length of "String"
substring(String,StartPos,Length)	Returns "Length" characters of "String" starting from "StartPos"
replace(StringSource,StringFindWhat,StringReplaceWith)	Replaces all the occurrences of "StringFindWhat" in "StringSource" with the string "StringReplaceWith"
left(String,Length)	Returns the first "Length" characters of "String"
right(String,Length)	Returns the last "Length" characters of "String"
concat(String1,String2)	Returns the concatenation of the two strings "String1" and "String2"
IndexOf(StringToSearch,StringToFind,StartIndex)	Returns the index of the first occurrence of "StringToFind" within "StringToSearch" starting from "StartIndex". A value of -1 is returned if the search string is not found.
trim(StringToTrim,CharactersToRemove)	Removes the characters listed in "CharactersToRemove" from the beginning and the end of "StringToTrim"
trimleft(StringToTrim,CharactersToRemove)	Removes the characters listed in "CharactersToRemove" from the beginning of "StringToTrim"
trimright(StringToTrim,CharactersToRemove)	Removes the characters listed in "CharactersToRemove" from the end of "StringToTrim"
tolower(String)	Converts "String" into lowercase
toupper(String)	Converts "String" into uppercase
tostring(Number)	Converts "Number" into a string

Examples

SYMBOL	DESCRIPTION	EXAMPLE	RESULT
like	Wildcard string compare	See below.	See below.
len	String length	x=len("Hello World")	11
substring	Substring extraction	x=substring("Hello World",0,5)	Hello
replace	Substring replacement	x=replace("Status busy","busy","ready")	Status ready
left	Left substring	x=left("Hello World",5)	Hello
right	Right substring	x=right("Hello World",5)	World
concat	String concatenation	x=concat("Hello ", "World")	Hello World
IndexOf	String search	x=indexof("Hello World","World",0)	6

SYMBOL	DESCRIPTION	EXAMPLE	RESULT
trim	Trim left and right	x=trim("Hello World","Hdl")	ello Wor
trimleft	Trim left	x=trimleft("Hello World","Hdl")	ello World
trimright	Trim right	x=trimright("Hello World","Hdl")	Hello Wor
tolower	Lower case cast	x=tolower("Hello World","Hdl")	hello world
toupper	Upper case conversion	x=toupper("Hello World","Hdl")	HELLO WORLD
tostring	Type conversion	See below.	See below.

Note: When a string value, which can be converted into a number, is passed as a numeric parameter for a string function, then an automatic conversion is attempted. For example, the expression:

x=Left("Hello World","4")

is a valid expression.

If the automatic conversion is not possible, then the expression result will have bad quality. See the data type conversion section above for additional details.

Note that is also possible to pass a number to a string parameter (refer to the "Data Type Conversion" section above). For example:
x=len(35) will return 2

Note also that the string functions will return bad quality when the parameters are non-meaningful (e.g. a negative length or a negative starting position).

Function: like(StringSource,StringCompare,BooleanMatchCase)
The like function is used to perform string comparison; it searches for a pattern inside a string and returns a value that indicates whether the pattern is contained in the string or not. The "like" operator accepts three parameters:

1. The first parameter is the string in which you want to perform the search.
2. The second parameter is the pattern that you want to find.
3. The third parameter is a Boolean parameter that allows you to specify whether the search must be case-sensitive or not (1 meaning case-sensitive, and 0 meaning case-insensitive).

A string can be a variable of type string, a process point (TAG) of type string, an array of strings (meaning OPC tag of type array of string), or a constant value of type string, or a numeric value.
An example of a constant string is:

"This is a string"

Notice that the " (quotation mark) indicates the beginning of the string, and the " (quotation mark) indicates the end of the string.

For the "like" operator: "string" equals the string to search in; "pattern" equals the string to search for (can include wildcards); nonzero for case-sensitive search; zero for case-insensitive search. String syntax is "string".

You can use these special symbols in pattern matches to match a single character in the source string:

- ? Accepts any single character.
- # Accept a single character if it is digit (0-9).
- [charlist] Accepts a single character if it is part of the characters list.
- [!charlist] Accept a single character if it is not part of the characters list.

Note To match the special characters left bracket (**[**), question mark (**?**), number sign (**#**), and

asterisk (*****), enclose them in brackets. The right bracket (**]**) cannot be used within a group to match itself, but it can be used outside a group as an individual character.
By using a hyphen (**-**) to separate the upper and lower bounds of the range, *charlist* can specify a range of characters. For example, [A-Z] results in a match if the corresponding character position in *string* contains any uppercase letters in the range A-Z. Multiple ranges are included within the brackets without delimiters.

Other important rules for pattern matching include the following:

- An exclamation point (!) at the beginning of charlist means that a match is made if any character except the characters in charlist is found in string. When used outside brackets, the exclamation point matches itself.
- A hyphen (-) can appear either at the beginning (after an exclamation point if one is used) or at the end of charlist to match itself. In any other location, the hyphen is used to identify a range of characters.
- When a range of characters is specified, they must appear in ascending sort order (from lowest to highest). [A-Z] is a valid pattern, but [Z-A] is not.

The character sequence [] is considered a zero-length string ("").

You can also use the * (asterisk) to match zero or more characters in the string.
It is important to note that the "like" function performs a character-by-character comparison of the source and the compare strings. If these strings have different length, then the comparison will not be possible and the function will return FALSE. The only exception to the aforementioned rule is through the use of the * (asterisk), which can take the place of zero or more characters.

The following is an example of the "like" function:

```
Like(~~var_user_name~~,"mark",1)
```

The above expression will look inside the string variable ~~var_user_name~~ and will return true if it contains the exact string "mark."

If the variable ~~var_user_name~~ contains the string "MaRk," then the above expression will return false because we have set the comparison to be case-sensitive (the third parameter is 1).

If we write the following expression:

```
Like(~~var_user_name~~,"mark",0)
```

Then it will return true even if ~~var_user_name~~ contains the string "MaRk," because we have set the comparison to be case-insensitive.

If we write the following expression:

```
Like(~~var_user_name~~,"mark",0)
```

And the variable ~~var_user_name~~ contains the string "mark twain," then the expression will return false because the two strings do no match exactly.

As mentioned above, you can use wildcard characters to compare partial strings.

The following table contains examples. The first parameter is normally a variable, but for this purpose it has been resolved.

FUNCTION	RETURN	COMMENTS
like("mark","mark",1)	1	The two strings are exactly the same; thus the expression returns true.
like("mark","Mark",1)	0	We have set the comparison to be case-sensitive, and the case of the two strings does not match; thus the expression returns false.
like("mark","Mark",0)	1	The two strings have different cases, but the expression returns true because we have set the string comparison to case-insensitive.
like("mark","ma*",1)	1	The constant part of the string is a match. The * syntax causes the rest of the string to automatically match.
like("mark","ma??",1)	1	The constant part of the string matches. The ? allows any character to exist in that location.
like("mark","ma?",1)	0	The constant part of the string matches. The ? allows any

FUNCTION	RETURN	COMMENTS
		character to exist in that location. In this case there are not enough ? to match the length of the string. Thus the resolution is "mar," which does not match "mark."
like("mark","m?rk",1)	1	The ? can be used anywhere in a string. This allows "m?rk" to be resolved "mark."
like("mark","m[abc]rk",0)	1	In this example, the square brackets enclose a list of characters that can be used to match the second character in the source string.
like("mark","ma[def]k",0)	0	Similar to the above example, but in this case we are trying to match the third character.
like("mark","[a-z]ark",0)	1	In this example, we are accepting for the first character any character between “a” and “z”.
like("mark","[a-z]a*",0)	1	This is similar to the above example, but we are accepting anything after the first two characters.
like("mark","[a-z]a",0)	0	This is similar to the above example, but in this case the like function returns zero because there are characters left after the first two characters have been compared. In other words, the two strings have different lengths, and there are no asterisks to accept for the extra characters.
like("mark","m[!bc]rk",0)	1	This will return 1 because we have prefixed the character array with a exclamation mark, and the second character does not belong to the character array.
like("mark2","mark#",0)	1	This will return 1 because the last character is a number.

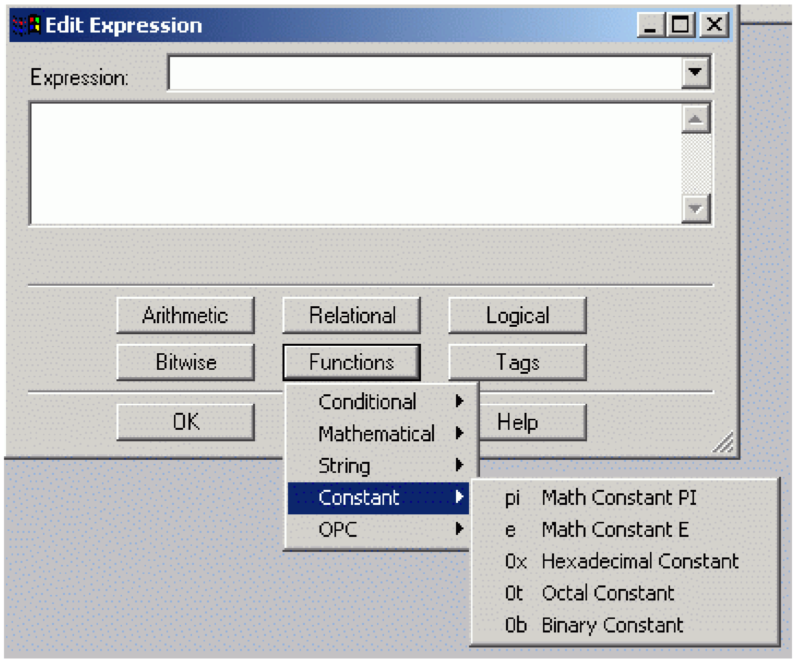
Function: tostring(number)
The **tostring** option on the **Functions** menu of the **Expression Editor** accepts anything that can be interpreted as a valid number in parentheses and converts it into a string.

Example Expressions Type Conversion

EXPRESSION	RESULT
x="The value is " + tostring({{gfwsim.ramp.float}}) + " Watt"	“The value is 543.2345152 Watt”

Constant

The **Constant** functions are shown in the figure below.



Constant Functions

The **Functions** menu of the **Expression Editor** supports constant values, including pi, *e,* hexadecimal, octal, and binary formats.

Examples

SYMBOL	DESCRIPTION	EXAMPLE	RESULT
pi	Math constant pi	x=pi	3.14159265358979323846
e	Math constant *e*	x=e	2.71828182845904523540
0x	Hexadecimal constant	x=0x11	17
0t	Octal constant	x=0t11	9
0b	Binary constant	x=0b11	3

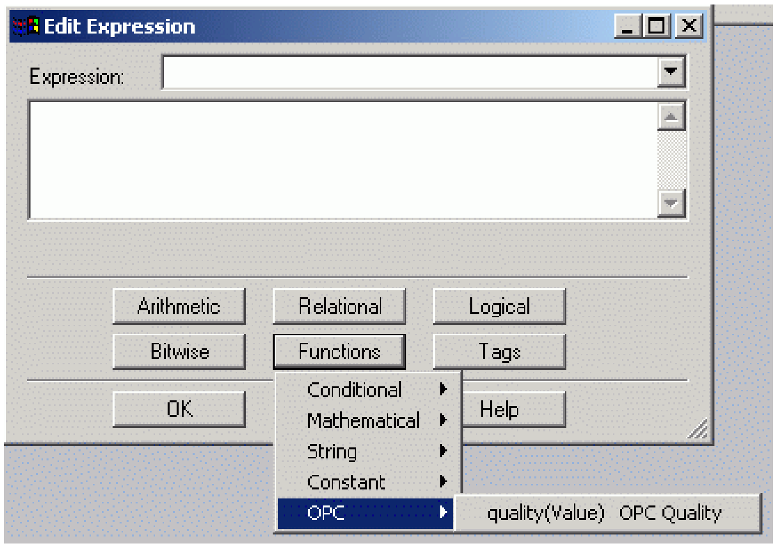
Note: The numeric constants "pi" and "e" are approximations of their real values. The rounded values are shown in the table above. Using rounded values for the mathematical constants "pi" and "e" may produce in unexpected results. For example, since the constant pi, presented here, actually is the rounded number pi, the expression such as tan(PI*N/2), where N is an odd number, is considered here as a valid expression.
Interpreting and Translating Constants

The examples below show how values are calculated for the hexadecimal, octal, and binary constants. The Expression Editor conveniently inserts the 0x and 0t and 0b prefixes for you so do not have to recall them.

- **Hexadecimal:** 0x20A = 2 * (16^2) + 0 * (16^1) + 10 * (16^0) = 2*256 + 0*16 + 10 * 1 = 512 + 0 + 10 = 522
- **Octal:** 0t36= 3 * (7^1) + 6 *(7^0) = 3* 7 + 6* 1= 21 + 6 = 27
- **Binary:** 0b110 = 1 * (2^2) + 1 * (2^1) + 0 * (2^ 0) = 1 * 4 + 1 * 2 + 0 * 1 = 4+2+0 = 6

OPC Quality

The **OPC Quality** function is shown in the figure below.



OPC Quality Function

The **quality** option on the **Functions** menu of the **Expression Editor** is used to evaluate the quality of an OPC tag or an expression.

The following general syntax is used for quality expressions:
x=quality(expression)

Note: The "(expression)" can also be a simple expression composed of a single tag.

The **quality** function returns the OPC quality of the string between parentheses as one of the following results:

- 192: quality is GOOD
- 64: quality UNCERTAIN
- 0: quality BAD

Note: The OPC Foundation establishes the value ranges for quality. There are actually varying degrees of quality:

- GOOD: 192-252
- UNCERTAIN: 64-191
- BAD: 0-63

For more information, refer to the *OPC Data Access Custom Interface Standard* available for download at the OPC Foundation's Web site, www.opcfoundation.org/.

Example Quality Expression

EXPRESSION	RESULT
x=quality({{Smar.Simulator.1\SimulatePLC.PumpStatus}})	192 (Quality GOOD)

The quality of an expression is determined through the evaluation of each single tag in the expression. Thus, if you have multiple tags in an expression (and each tag has a different quality), the result of the expression (i.e. 192 [GOOD], 64 [BAD], or 0 [UNCERTAIN]) corresponds to the quality of the tag with the lowest quality. If an expression contains a conditional statement (e.g. if, then, or else), then the result of the expression is affected only by the quality of the branch being executed.

Consider the following sample expression:
x= if (quality({{Tag1}}) == 192, {{Tag1}}, {{Tag2}})
This expression can be read as follows:

"If the quality of Tag1 is GOOD (i.e. 192), then the expression result (x) is the value of Tag1. In all other cases (i.e. the quality of Tag1 is UNCERTAIN or BAD), the expression result (x) is the value of Tag2."

We can calculate the results for this expression using different qualities for Tag1 and Tag2, as shown in the figure below.

CASE	TAG1 QUALITY	TAG2 QUALITY	RESULT	RESULT QUALITY
1	GOOD	GOOD	Tag1	192 (GOOD)
2	GOOD	UNCERTAIN	Tag1	192 (GOOD)
3	GOOD	BAD	Tag1	192 (GOOD)
4	UNCERTAIN	GOOD	Tag2	192 (GOOD)
5	UNCERTAIN	UNCERTAIN	Tag2	64 (UNCERTAIN)
6	UNCERTAIN	BAD	Tag2	0 (BAD)
7	BAD	GOOD	Tag2	192 (GOOD)
8	BAD	UNCERTAIN	Tag2	64 (UNCERTAIN)
9	BAD	BAD	Tag2	0 (BAD)

In cases 1-3 above, the quality of Tag1 is GOOD, and therefore the result of the expression is GOOD. Thus, the result of the expression is not affected by the quality of Tag2 (the "else" branch of the expression), which is ignored.
In cases 4-6, the quality of Tag1 is UNCERTAIN, and therefore the result of the expression is the quality of Tag2.
In cases 7-9, the quality of Tag1 is BAD, and therefore the result of the expression is the quality of Tag2.

Note: The "quality()" function returns a value that represents the quality of the expression within the parentheses but is always GOOD_QUALITY. For example, if Tag1 is BAD_QUALITY then the expression "x=quality({{Tag1}})" will return 0 with GOOD_QUALITY.
The result of an expression is the minimum quality of the evaluated tag in the expression and is affected only by the quality of the conditional (if, then, or else) branch that is executed.

Consider the following sample expression:
x= if ({{TAG_01}}>0,{{TAG_02}},{{TAG_03}})

This expression can be read as follows:
"If the value of TAG_01 is greater than 0, then the expression result (x) is TAG_02. If the value of TAG_01 is less than or equal to 0, then the expression result (x) is TAG_03."
Let's assume that the following values and qualities for these tags:
TAG_01=5 with quality GOOD
TAG_02=6 with quality UNCERTAIN
TAG_03=7 with quality BAD

Because the value of TAG_01 is 5 (greater than 0), the expression result is TAG_02. Thus, the final expression result is 6, and the final expression quality is UNCERTAIN.

Note: The quality of an invalid expression or number is always BAD.

Tags

The menu options available under the **Tags** button of the Expression Editor vary with each type of application and may include the following:

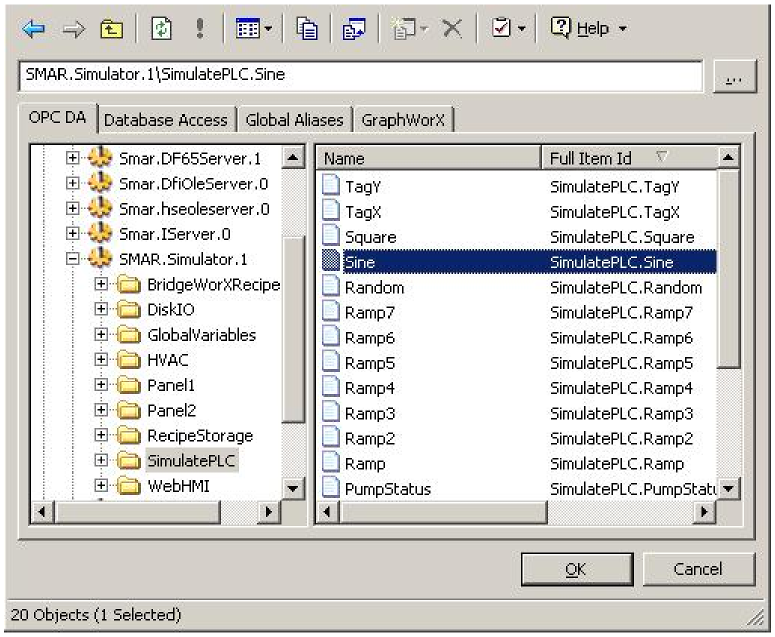
- OPC Tags
- Local and Global Aliases
- Variables
- Alarm Filters

NOTE
Some of these options may not be available in some applications. For more information about the available options, please refer to the application's help documentation.

OPC Tags
An **OPC tag,** or data point, is a data connection between a client and an OPC server. OPC tags can be used in expressions when the tag is embedded between double brackets, as shown below:
{{tag_name**}}**

Example:
x={{Smar.Simulator.1\SimulatePLC.PumpSpeed}}

You can use the Unified Browser, shown below, to select OPC Alarm and Event (AE), Data Access (DA), and Historical Data Access (HDA) tags to include in your expressions.



SMAR.Simulator.1\SimulatePLC.Sine

- Smar.DF65Server.1
- Smar.DfiOleServer.0
- Smar.hseoleserver.0
- Smar.IServer.0
- SMAR.Simulator.1
  - BridgeWorXRecipe
  - DiskIO
  - GlobalVariables
  - HVAC
  - Panel1
  - Panel2
  - RecipeStorage
  - SimulatePLC
  - WebHMI

Name	Full Item Id
TagY	SimulatePLC.TagY
TagX	SimulatePLC.TagX
Square	SimulatePLC.Square
Sine	SimulatePLC.Sine
Random	SimulatePLC.Random
Ramp7	SimulatePLC.Ramp7
Ramp6	SimulatePLC.Ramp6
Ramp5	SimulatePLC.Ramp5
Ramp4	SimulatePLC.Ramp4
Ramp3	SimulatePLC.Ramp3
Ramp2	SimulatePLC.Ramp2
Ramp	SimulatePLC.Ramp
PumpStatus	SimulatePLC.PumpStatu

20 Objects (1 Selected)

Selecting OPC Tags from the Unified Browser

Local and Global Aliases
An **alias** is a string that represents or describes an object or data point in a display. Both local and global aliases can be used in expressions.

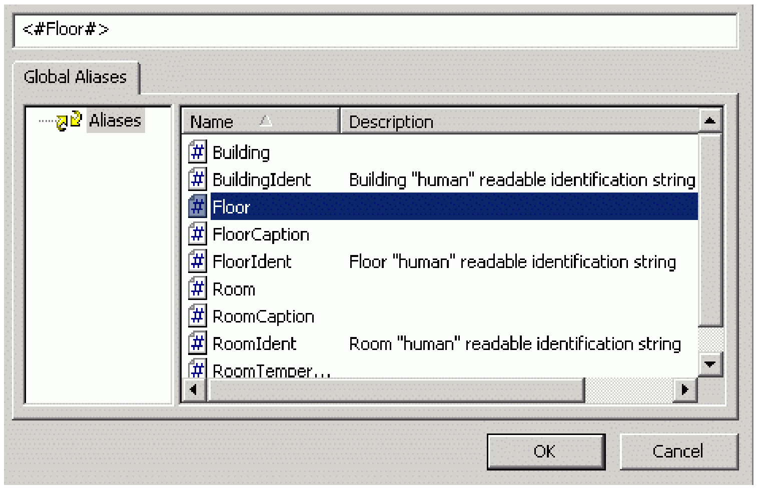
Local Aliases
For local aliases within the expression, use the following syntax:
<<local_alias_name**>>**

Example:
x=<<TankLevel>>

Global Aliases
For global aliases within the expression, use the following syntax:
<#global_alias_name**#>**

Example:
x=<#RoomTemperature#>

Select a global alias from the Unified Browser, which includes all global aliases in the global alias database. This eliminates the need to manually type in the alias name. All global aliases that are configured in the Global Alias Engine Configurator are conveniently available to choose from inside the browser. The tree control of the Global Alias Engine Configurator is mimicked in the tree control of the Global Alias Browser. Select a global alias by double-clicking the alias name (e.g. "Floor" in the figure below). The alias name appears at the top of the browser, which automatically adds the <# and #> delimiters to the alias name. Click the **OK** button.



<#Floor#>

Name	Description
Building	
BuildingIdent	Building "human" readable identification string
Floor	
FloorCaption	
FloorIdent	Floor "human" readable identification string
Room	
RoomCaption	
RoomIdent	Room "human" readable identification string
RoomTemper...	

Selecting a Global Alias From the Unified Browser

Variables
Variables can be used in expressions. How the variable needs to be referred depend on the type of variable. A local variable can be used in expressions when the variable is embedded between double tildes.

Local Variables
For local variables within the expression, use the following syntax:
~~local_variable_name~~

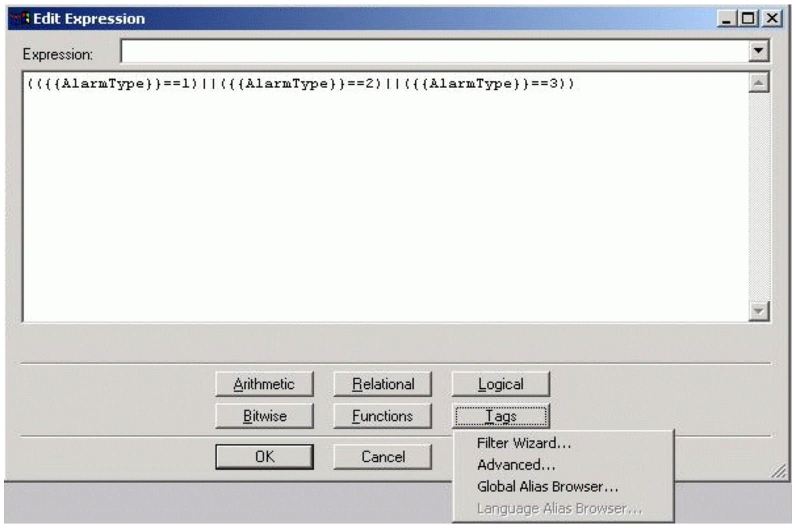
Example:
x=~~Setpoint~~

Simulation Variables
For simulation variables within the expression, use the following syntax:
{{simulation_variable_name**}}**

x={{gfwsim.random.long}}

Alarm Filters

The **Expression Editor** dialog box, shown in the figure below, can also be used to edit alarm filters (see the AlarmWorX help documentation for additional information about filters). The Expression Editor provides a Filter Wizard and an Alarm Tag list to help you create simple alarm filters. If you want to customize your alarm filters, you can use the other functions in the Expression Editor to set up your alarm filters manually.



```
((({AlarmType})==1)||(({AlarmType})==2)||(({AlarmType})==3))
```

Editing Alarm Filters Using the Expression Editor

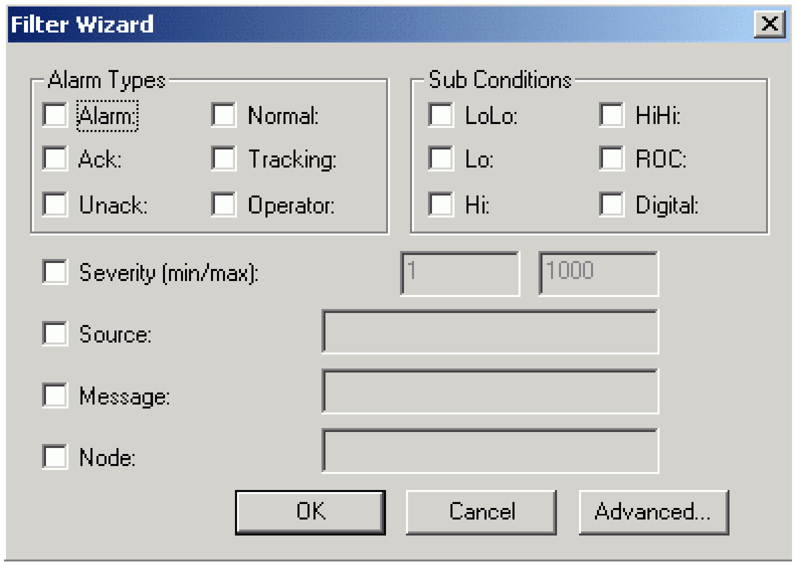
Filter Wizard

The **Filter Wizard,** shown in the figure below, allows you to choose from the following items to enter in your expression. Select one or more items, and then click **OK.** The filter string is automatically inserted into the **Edit Expression** dialog box.

- **Alarm Types:** Alarm, Ack, Unack, Normal, Tracking, and Operator
- **Subconditions:** LoLo, Lo, Hi, HiHi, ROC, and Digital
- **Severity:** This is the OPC-defined value for alarm priority. The valid OPC severity range is 1 (lowest) to 1000 (highest). The first number is the low limit, and the second number is the high limit.
- **Source:** The tag name.
- **Message:** Description of the tag.
- **Node:** Name of the PC from which the tag originates.

Clicking the **Advanced** button opens the Expression Editor.

Note: Alarm Types are logically "or" with Alarm Types. Subconditions are logically "or" with Subconditions. Alarm Types, Subconditions, Severity, Source, Message, and Node are logically combined ("and") to create the expression string.

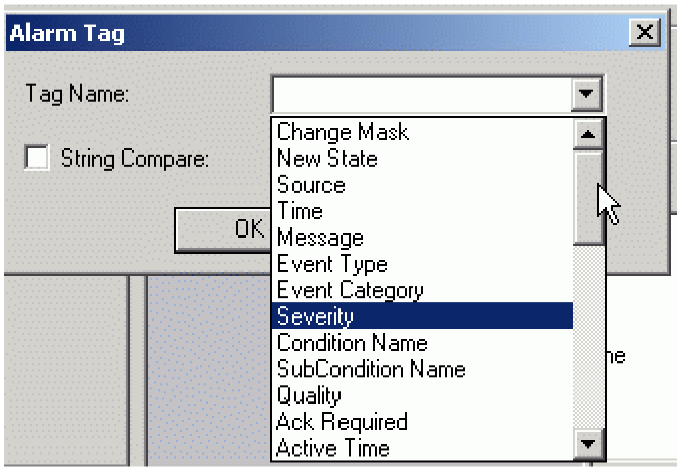


Filter Wizard

Selecting Alarm Attributes

Selecting **Advanced** from the **Tags** menu of the Expression Editor opens the **Alarm Tag** list, shown in the figure below, which allows you to choose alarm attributes for your alarm filter. Select the attribute that you want to include in the filter expression and click **OK.**

Note: You can also perform a wildcard string compare using the "like" operator syntax. Check the **String Compare** check box and type the desired string in the text field.



Alarm Attributes List

There are two additional attributes available for use in filtering: **Alarm Type** and **Current Time.** The Alarm Type attribute allows you to filter alarms according to ALARM 1, ACK 2, UNACK 3, OPER 4, TRACK 5 or NORM 6. For example, you can set up a filter with the condition:

X = {{AlarmType}}

If the **Alarm Type** is true, then the alarms are displayed. If they are false then, the alarms are not displayed.

The **Current Time** attribute allows you to filter according to the current time. Only alarms occurring around the current time will be displayed.

Example Alarm Filters

EXPRESSION	RESULT
X = {{Severity}} > 500.	Only alarm messages with a severity greater than 500 will be visible.
X = Like({{Source}}, "Tag",0)	Only messages with the tag in the source name will be displayed.
X = Like({{Message}}, "Boiler Pump not*",0)	Only messages with a description beginning with “Boiler Pump not” will be displayed.
X = Like({{Node}}, "Alpha",0)	Only messages from Node Alpha will be displayed.
X = Like({{Attribute1}}, "Plant Area 1",0) X = {{Attribute2}} == 40.	Tag values Attribute1 through Attribute20 are server specific. They can be strings, numeric values, or arrays. The most common array usage is for Areas. When dealing with areas, use the Like function.
X = 1.	Filter displays all messages.
X = 0.	Filter does not display any messages.

All filters resolve to TRUE or FALSE. All nonzero values resolve to TRUE.

For more information, please see the AlarmWorX Server documentation.