About This Manual

How to Use This Manual Set

The *Getting Started* guide contains installation instructions and a basic introduction to Lookout*Direct* features and functionality. It includes general information to help acquaint you with the important elements of Lookout*Direct* along with tutorial exercises to introduce you to the basics of building a Lookout*Direct* process.

The *Object Reference Manual* is in Portable Document Format (PDF) and describes the Lookout*Direct* object set. It is available on the Lookout*Direct* CD in the `documentation` directory. The Lookout*Direct* installation gives you the option of copying the PDF files into the `LookoutDirect/documentation` folder on your hard drive. To view these files, you must have Adobe Acrobat Reader 3.0 or later installed.
If you do not have Adobe Acrobat Reader installed, you can install it from the `LookoutDirect/documentation` directory or from the Adobe web site at *www.adobe.com.*

This *Developer's Manual* explains how to create control processes for your HMI/SCADA applications. This manual includes detailed explanations of various Lookout*Direct* features, functions, and services.

Document Conventions

The following document conventions are used in this manual.

»	Indicate the path for nested menu and command selections. Example: **Start » Programs » xxx**
💡	Indicates a tip that provides helpful information related to the current procedure or topic.
📝	Indicates a important supplementary information pertinent to the current procedure or topic.
⚠	Denotes a caution statement. Failure to follow the guidance provide in the caution statement could result in a loss of data or an interruption to a critical process being controlled or monitored by Lookout*Direct*.

bold face text	Denotes the name of dialog boxes, property sheet items, application features, and menu commands that you select as part of a procedure.
italic	Denotes references to dialog boxes, property sheet items, application features, and menu commands that are not part of the current procedure, or key concepts.
`monospace`	Denotes characters that you enter from the keyboard, code/syntax examples. The monospace font is also used to express proper names of disk drives, paths, directories, programs, subprograms, subroutines, device names, functions, operations, variables, filenames and extensions.

Technical Support

Phone

Contact Automation Direct's technical support department toll free at 1-770-844-4200. Technical support is available weekdays from 9:00 a.m through 6:00 p.m. EST.

World Wide Web

Visit the Automation Direct website at *www.automationdirect.com* and click on *Technical Support* to access a wide variety of technical support assets including print-on-demand documentation and software downloads.

1 Expressions

This chapter explains the features and uses of Lookout*Direct* expressions, which can act as variables, capable of flexible, real-time math statements, condition testing, and other complex operations.

Lookout*Direct* expressions can use spreadsheet-style formulas with a mixture of constants and variable signals from objects. They can be short and simple, or extremely complicated with several signal inputs, function calls, and multiple levels of parentheses.

A single expression can incorporate any number of numeric, logical, and text signals within its calculation. However, the *result* of the expression can be only one of three types: numeric, logical, or text. The outermost function or operator in the expression returns a variable type that determines the overall signal type of the expression.

While an expression produces a single type of data, most Lookout*Direct* objects interpret data according to the data type that object needs for a particular data member. You can connect a logical output to a numeric input, and the arriving data will be interpreted as a 0 or a 1. Refer to the *Data Polymorphism* section of this chapter for detailed information on how Lookout*Direct* interprets data without respect to the original type.

Simple expressions consist of a single value. Simple expressions do not contain any modification, manipulation, math, or logic (for example, a single object with a name like `Pot1`). When you add any functionality to a simple expression, you create a complex expression (for example, `Pot1 > 33`).

The following examples are typical expressions. Depending on your system requirements, your expressions might be much more involved.

True

Always returns the value true (on).

Pot1

Returns the current value [the `(intrinsic)` data member] of Pot1.

879.03

Always returns the value 879.03.

1:04:33

Returns the value 0.0448264. If formatted in hours, minutes, and seconds, it is displayed as 1:04:33.

Pressure * 2.31

Multiplies Pressure by 2.31.

(Switch1 or Switch2) and TankLevel > 65.2

If Switch1 or Switch2 is true, and TankLevel is greater than 65.2, return true. Otherwise, return false.

Pressure >= Setpoint and !ReliefValve and TimeOccuring > 0:30

If Pressure is greater than or equal to Setpoint, and ReliefValve is false, and TimeOccuring is greater than 30 seconds, return true. Otherwise, return false.

if(PLC1.AutoPos, AutoTemp, if(PLC1.ManPos, ManualTemp,0))

If the alias member AutoPos of PLC1 is true, return the value of AutoTemp. If the alias member ManPos of PLC1 is true, return the value of ManualTemp. Otherwise, return the value 0.

tif(HOA=1,“Hand”,tif(HOA=2,“Off”,“Auto”))

`tif` is the text version of the `if` function. If the HOA switch is at position 1, return `Hand`. If the HOA switch is at position 2, return `Off`. Otherwise, return `Auto`.

You can accomplish much the same thing with the following `tchoose` expression as you can with the previous `tif` expression.

tchoose(HOA,"hand","Off","Auto")

If the HOA switch is at position 1, return `Hand`. If the HOA switch is at position 2, return `Off`. If the HOA switch is at position 3, return `Auto`.

Make sure that your expressions do not attempt to calculate illegal operations, such as dividing by zero or finding the arc cosine of a number greater than 1. If a signal can assume a value that could cause Lookout*Direct* to attempt an illegal mathematical operation, you should specifically test for that condition in the expression. Expressions trap illegal mathematical operations and generate alarms in the Math alarm area. The alarms do not reset until you correct the expression. The following list includes some of the illegal conditions that Lookout*Direct* traps:

- attempting to divide by zero

- taking the square root of a negative number
- numeric underflow
- numeric overflow

Note If any value referenced in an expression changes when an event occurs, the expression automatically recalculates. This is the same event-driven concept that objects implement.

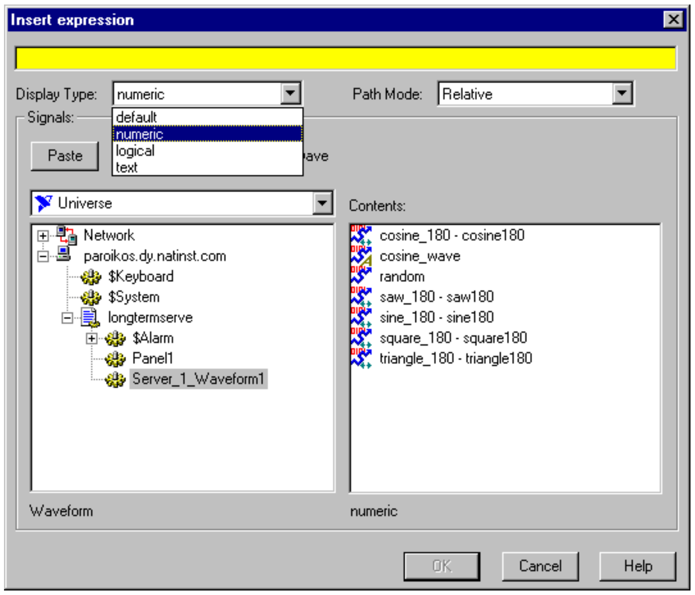
Creating Expressions

Because Lookout*Direct* uses expressions in many places, you can create expressions as independent objects, parameters in objects, connections to object data members, or you can insert expressions on control panels.

Through dialog boxes, you create expressions during process development. There are three types of data entry fields in Lookout*Direct* dialog boxes—white, yellow, and green. White data entry fields accept only constant values, yellow data entry fields accept expressions, and green fields accept URLs.

Data Polymorphism

Lookout*Direct* data is not strongly typed, which means that it does not need to be interpreted as the type it was originally cast in. Data polymorphism in Lookout*Direct* means that data of one type is interpreted appropriately when connected to an input of another type. You can select the data type you want an expression to be displayed as in the **Display Type** field, shown in the following dialog box.



The following table describes how data types are used by inputs of a different type.

Table 1-1. Data Type Conversion

Data Type	**Interpreted as Numeric**	**Interpreted as Text**	**Interpreted as Logical**
Logical	0 or 1	ON or OFF	—

Table 1-1. Data Type Conversion (Continued)

Data Type	Interpreted as Numeric	Interpreted as Text	Interpreted as Logical
Numeric	—	digits of the number	`0` displays as OFF Any value other than `0` displays as ON
Text	Appear as digits if the string consists only of digits in a valid LookoutDirect display format, such as a decimal number or a scientific expression. In a time format, such as `10:05:30`, the text is interpreted as a number (the fraction of one day represented by the time quantity) in scientific notation. If the text string does not consist of digits in a valid LookoutDirect format, the text is interpreted as a 0, with the exception of `ON` or `TRUE`, which display as 1. The text strings `OFF` or `FALSE` are interpreted as 0, by default.	—	Interpreted as a 0 or as OFF, except when the text string consists of a `1`, `On`, or `True`, all of which are interpreted as ON. By default, the text strings `0`, `Off`, or `False` are interpreted as OFF.

After you have placed an expression on your panel, you cannot change the data type in that expression. You have to delete that expression and place a new one if you want to change data types.

Expressions used as parameters in Lookout*Direct* objects do not permit you to select the data type because the object interprets input according to the data type required by that object. Making Lookout*Direct* data polymorphic permits you to use output of a data type different from that required by an object, as long as the output can be interpreted meaningfully by the object.

Polymorphic data types affect the DataTable object. In versions earlier than Lookout*Direct* 4, the DataTable used data members such as `A1.logical` to set the type of data in a particular cell.

Note The Lookout*Direct* 3.*xx* DataTable data members are preserved in Lookout*Direct* 4 for the sake of compatibility. It is not necessary to use the data-typed data members to simulate polymorphic data. To achieve this compatibility, however, Lookout*Direct* DataTable data is still strongly typed. You must input numeric data to DataTable numeric data members, logical data to DataTable logical data members, and so on.

Path Names in Lookout*Direct*

In versions of Lookout*Direct* prior to Lookout*Direct* 4, you could only have one process running in any instance of Lookout*Direct*, and there were no folders in a process to organize objects. Networking was done through DDE, which used specific paths. In Lookout*Direct* 4, the relationship between objects running in different processes on different computers is far more flexible and potentially complicated.

When you insert an expression on a Lookout*Direct* panel or make a connection to or between data members, you might be displaying a value that comes from the process containing that panel. You might also need to display a value that comes from another process running on the same computer or from a process running on some other computer in your network.

A path can be *absolute*, starting with a computer's fully qualified network name and leading through nested directories and subdirectories to a single object; or *relative*, instructing the computer to search for an object in some directory defined not from the computer down, but relative to the currently active folder.

To provide maximum programming flexibility, you can use a number of different levels of relativity, or *path relativity modes*, in setting the path to the data when you create an expression. These modes operate very much like other relative paths used in DOS operations, spreadsheet cells, and Web page addressing.

You can set a path for each individual object or connection, using a different level of relativity. These path relativity modes include

- Relative (the Lookout*Direct* default)
- Process Relative
- Computer Relative
- Absolute

The level of path relativity you choose makes a difference in how your process operates when copied or moved to another computer. For instance, suppose you have a client process running on computer Alfred that refers to objects in a server process also running on computer Alfred. If you run that client process on computer Bert, will it refer to objects on computer Alfred or computer Bert?

Setting the level of path relativity is how you can control which objects your processes refer to. The key point to remember about path relativity modes in Lookout*Direct* is that the modes are distinguished by the path prefix.

In *Relative* mode, the form of the path is

`object`

or

`folder\object`

without prefix.

Relative mode is the default mode in Lookout*Direct* 4. All paths are relative to the folder or process that contains the expression you are creating or the object you are connecting to. Use Relative mode when you connect objects that should be grouped together and that should be copied or moved as a group.

Connections between objects in a folder should use Relative mode so that they can be copied and moved elsewhere while maintaining the same relationship with each other.

Process Relative mode is indicated by a single prefixed backslash (\) followed by a folder or object name, as in

`\folder\...folder\object`

or

`\object`

Use Process Relative mode when you want to make a connection between two objects in different folders in the same process.

Computer Relative mode is indicated by a prefix consisting of two backslashes, a period, and a backslash (`\\.\`) followed by a process name, as in

`\\.\process\folder\...folder\object`

Use Computer Relative mode when you want to reference other processes running on the same computer.

For instance, if you had a process called `Station_Control` that called a number of other processes by name on computer Alfred, you could use that same `Station_Control` process on computer Bert to refer to the identically named processes running on computer Bert.

Absolute mode is indicated by two backslashes (\\) followed by a complete network path, as in

`\\computer.place.com\process\folder\...folder\object`

Absolute mode is, in effect, a URL. Use Absolute mode when you want to refer to a particular process on a particular computer. Any references to a PLC, for instance, should be in Absolute mode so as to always refer to the computer physically connected to that PLC—no matter where the process making the reference is running.

Note Though Relative mode is the Lookout*Direct* default, when you use an object outside the immediate location of your expression, the path becomes more specific. The addition of two periods at the beginning of a path, for instance, can be used to specify the parent of the first folder listed, as in `..\folder\object`.

You should manually change a path mode to a level beyond that strictly necessary under your current circumstances if you want to ensure that your process will still work properly if you relocate it, completely or partially.

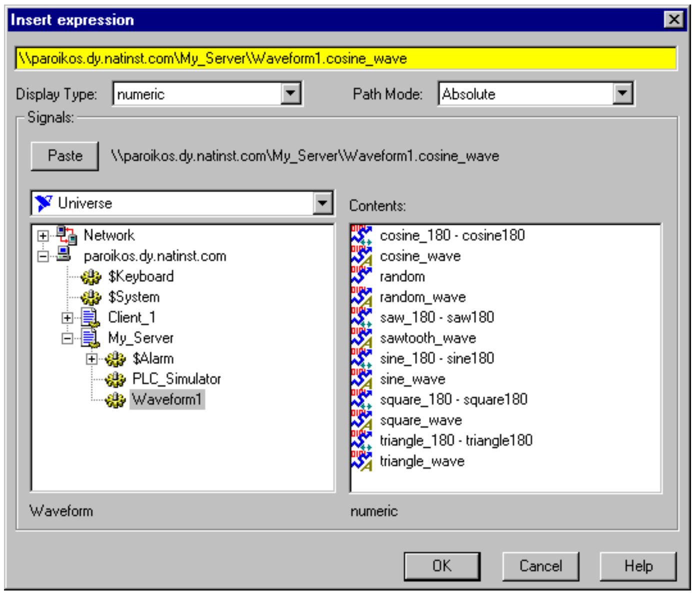
Expressions on Control Panels

With the **Insert»Expression** command, you graphically display the result of an expression on a control panel. However, this method does not create an object; therefore, it does not create an output signal that other expressions or objects can use. (There is no name associated with the expression.)

You can insert this same kind of expression by dragging and dropping the data member you want to display from the Lookout*Direct* Object Explorer.

Note When you insert an expression through the menu, you can explicitly choose the path relativity before you create the expression. If you insert an expression by dragging a data point from the Lookout*Direct* Object Explorer, the path will reflect the node you dragged the expression from. If you drag and drop an expression from the local node of Lookout*Direct* running on your computer, you will insert an expression with a relative path. If you drag an expression from a process under the network nodes for Lookout*Direct* processes (including the network node for your local computer), the expression will have an absolute path. Refer to the *Path Names in LookoutDirect* section for more information on Lookout*Direct* path modes.

The following illustration shows the **Insert Expression** dialog box you use to create or modify the path for a typical Lookout*Direct* expression.


Insert expression

\\paroikos.dy.natinst.com\My_Server\Waveform1.cosine_wave

Display Type: numeric

Path Mode: Absolute

Signals:

Paste \\paroikos.dy.natinst.com\My_Server\Waveform1.cosine_wave

Universe

- Network
- paroikos.dy.natinst.com
  - $Keyboard
  - $System
  - Client_1
  - My_Server
    - $Alarm
    - PLC_Simulator
    - Waveform1

Contents:

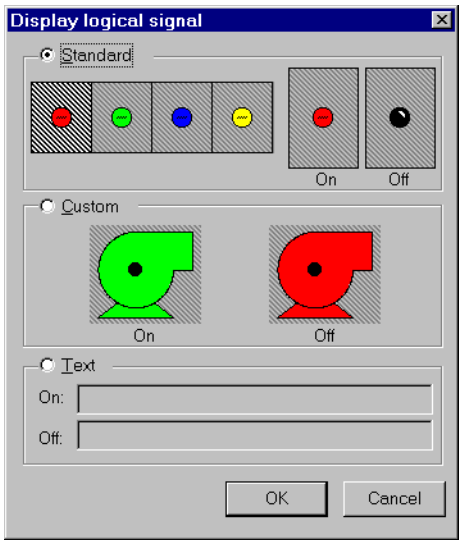
- cosine_180 - cosine180
- cosine_wave
- random
- random_wave
- saw_180 - saw180
- sawtooth_wave
- sine_180 - sine180
- sine_wave
- square_180 - square180
- square_wave
- triangle_180 - triangle180
- triangle_wave

Waveform

numeric

Notice that you can set the data type for the expression to display. After you have created an expression, you can no longer modify the data type, so be sure you select the correct type for your display.

The following illustrations show typical Lookout*Direct* dialog boxes to select the display properties for expressions.


Display logical signal

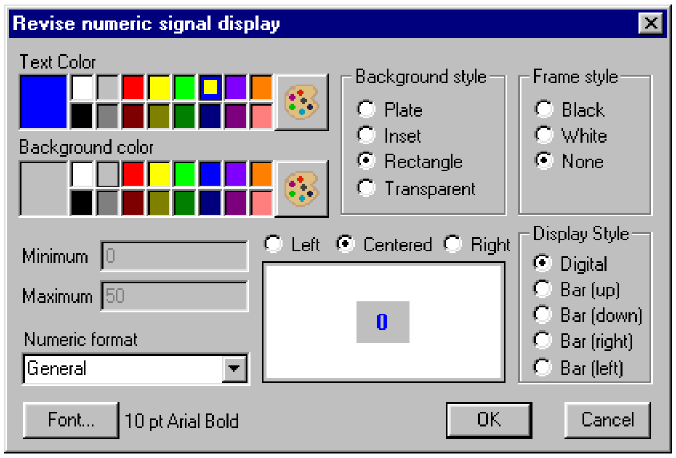
[✓ Standard]

Custom

Text

On:

Off:



Revise numeric signal display

Text Color

Background color

Background style: Plate / Inset / [✓ Rectangle] / Transparent

Frame style: Black / White / [✓ None]

Minimum: 0

Maximum: 50

Numeric format: General

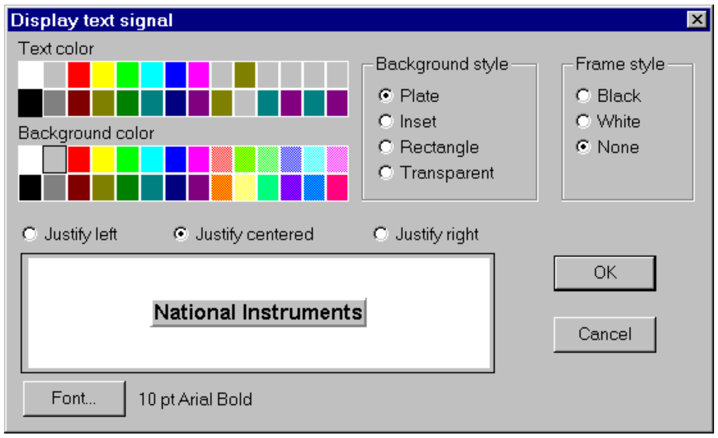
Left [✓ Centered] Right

0

Display Style: [✓ Digital] / Bar (up) / Bar (down) / Bar (right) / Bar (left)

Font... 10 pt Arial Bold

OK Cancel



Display text signal

Text color

Background color

Background style: [✓ Plate] / Inset / Rectangle / Transparent

Frame style: Black / White / [✓ None]

Justify left [✓ Justify centered] Justify right

National Instruments

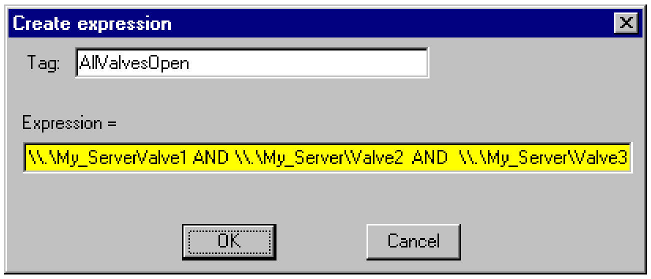
OK Cancel

Font... 10 pt Arial Bold

Expression Objects

With the **Object»Create** menu command, you can create an Expression object, or *named expression*. Like other object classes, the global (expression) object class requires a unique name.

Other expressions and/or objects can reference the output of an (expression) object. When you need to define a unique condition that your process uses multiple times, use an (expression) object. Instead of defining the same expression in many places, you can create it one time and use its name wherever the condition applies. For example, the name `AllValvesOpen` in the following **Create expression** dialog box is easier to reference than the long expression.


Create expression

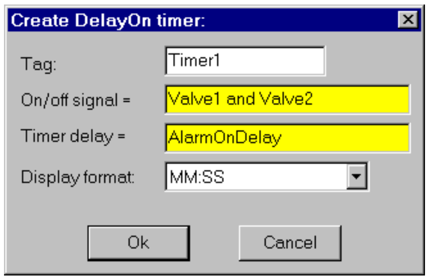
Tag: AllValvesOpen

Expression = \\.\My_Server\Valve1 AND \\.\My_Server\Valve2 AND \\.\My_Server\Valve3

You can also use the expression editor to assemble the long expression by right-clicking in the yellow field.

Expressions as Parameters

Many object classes accept expressions as parameters. When they do, the parameter expects the expression to return a certain signal type—numeric, logical, or text. For more information about object parameters and data members, refer to the specific object class definition in the online help.


Create DelayOn timer:

Tag: Timer1

On/off signal = Valve1 and Valve2

Timer delay = AlarmOnDelay

Display format: MM:SS

The parameter **On/off signal** expects a logical expression and **Timer delay** requires a numeric expression. Both `Valve1` and `Valve2` are names for switches. Connecting them with `and` produces a logical result and satisfies the type condition of the first parameter. `AlarmOnDelay` is the name of a potentiometer used to adjust the timer delay setpoint, which creates a numeric signal, satisfying the type condition of the second parameter.

Because the **Timer delay** parameter is an expression, we have many configuration possibilities, including the following:

- Enter a constant. In this case, the delay never changes.
- Enter the name of an output signal from another object such as a Pot. In this case, the operator adjusts.
- Enter a complex expression that automatically calculates the delay based on multiple inputs.

Similar configuration solutions exist for the **On/off signal** parameter.

When an expression parameter fields is yellow, you can get help building your expressions. For example, assume that you cannot remember whether the valve name in the previous example was `Valve1`, `Valve_1`, or `ValveOne`. If you position the cursor over the yellow expression field and right-click, the **Expression Editor** dialog box appears. Using this dialog box, you can select the proper name (`Valve1`) and paste it directly into the expression field. Click on **OK** and Lookout*Direct* writes the expression into the targeted parameter field.

Expressions as Connections

You can connect object data members with expressions. Writable data members accept expressions as inputs, much like parameters. To connect an expression to a data member, use the **Object»Edit Connections** menu command, or right-click on the object you want to connect to in the Object Explorer, and select **Edit Connections**. See Chapter 4, *Using LookoutDirect*, of the *Getting Started With LookoutDirect* manual for detailed information on connecting expressions to data members.

Edit/Insert Expression Dialog Box

You can insert an expression as a display element on a control panel, use it as a parameter for a Lookout*Direct* object, or use it as an input data member for a Lookout*Direct* object. When you use an expression as a display object, you can create or modify the expression using the **Insert Expression** dialog box. When you use an expression as a parameter or input to a data member, you can create or modify the expression using the **Edit Expression** dialog box.

These dialog boxes are almost identical, the only difference being that the **Insert Expression** dialog box contains a **Display Type** field that you use to select how the data expression will appear on your control panel. This field does not exist in the **Edit Expression** dialog box because an expression being used as a parameter or as an input to an object data member will be interpreted as the data type required for that input.

Note After you insert an expression for display, you cannot change the display type of the expression. The **Display Type** dialog box will be disabled if you open the expression to edit it. To change the data type, you have to insert a new expression.

You can access the **Insert Expression** dialog box in any of the following ways:

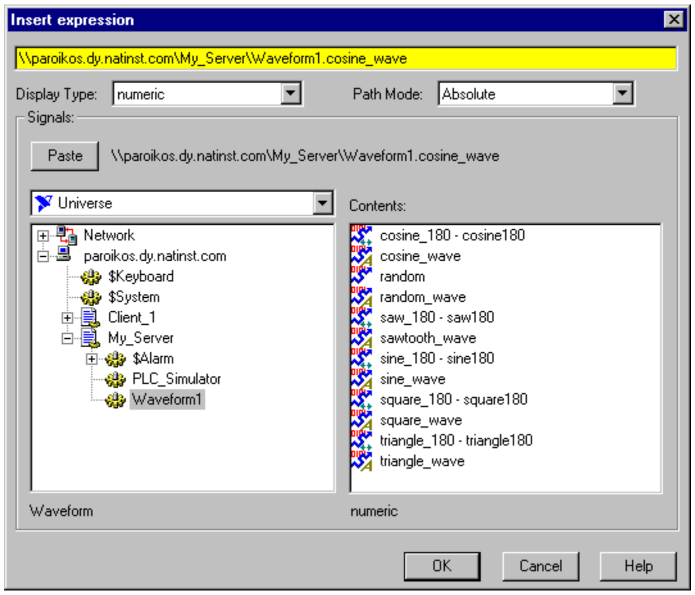
- Select **Insert»Expression** from the Lookout*Direct* menu.

- Click-drag an object from the Object Explorer.
- Right-click on an expression display on a control example and select **Object Properties**.
- Right-click in a control panel and select **Insert Expression**.

You can access the **Edit Expression** dialog box by right-clicking in any yellow expression field in any Lookout*Direct* dialog box.

Except for setting a display data type in the **Insert Expression** dialog box, you use the two dialog boxes in the same way. The following instructions and comments apply to both dialog boxes.

Figure 1-4 shows the **Insert expression** dialog box. You can enter your expression in the yellow expression field at the top of this dialog box.



Just below the yellow expression field are the **Display Type** and **Path Mode** selection boxes. The **Display Type** box selects the data type for your expression to be displayed with (if relevant), and the **Path Mode** box selects how detailed the path name of your expression will be.

The lower-left field lists all objects that generate readable signals under the root shown in the root window (under the **Paste** button). The lower-right field lists the readable data members (**Contents**) for the currently selected object. Notice that the dialog box indicates the *object class* selected in the object list (Waveform) and the *signal type* selected in the **Contents** list (numeric).

You can enter an expression directly or use the **Paste** button to select and insert object names in the expression field. As you select different objects from the listbox, the name to the right of the **Paste** button changes accordingly.

Some objects have multiple readable data members, such as a Modbus object. Lookout*Direct* concatenates the name, followed by a period and the selected data member.

After you combine the desired name and data member, click on the **Paste** button. Lookout*Direct* pastes the name into the expression field. Clicking on the **Paste** button a second time copies another instance of the name to the expression window—this time at the cursor.

Tip Instead of selecting a data member and then clicking on the **Paste** button, double-click on the data member. Lookout*Direct* automatically pastes the name into the expression window.

You can manually type or modify a name, a data member, or a mathematical function in the expression field at any time. Because you might have many hard-to-remember, defined objects, the navigation and selection listboxes serve as a quick reference for all of your previously defined objects.

Expression Syntax

In an expression, operators are instructions to perform an operation on a value or to combine values to form a new value. For example, a simple operator is the `/` symbol. It divides one value by another; for example, the formula (`5/2`) reads *five divided by two* and produces a result of 2.5. The five and two in the preceding example are operands (the `/` operator requires numeric operands).

White Space

You can use tabs and spaces between operators, functions, and function parameters in Lookout*Direct* to make expressions easier to read. This manual uses spaces between operators for added clarity.

Spaces and tabs are called *white space* characters because they provide space between parameters. This practice is used for the same reasons that spaces are used between words and paragraphs in a book—to achieve greater organization and clarity. Because Lookout*Direct* ignores white space characters, you can use white space characters to separate object names in an expression. However, you cannot embed white space characters within names or alias names.

Arithmetic Operators

The following operators perform basic arithmetic operations. They require numeric operands and produce numeric results.

Table 1-2. Arithmetic Operators

`+`	Addition
`–`	Subtraction
`*`	Multiplication
`/`	Division
`%`	Percentage (divides preceding value by 100)
`^`	Exponentiation
`–`	Negation (additive inverse of the following value)

Text Operator

The text operator is the ampersand character (`&`). An ampersand joins two text strings. For instance, the expression `“Call me“ & “Ishmael”` produces a text signal of `Call me Ishmael`. Typical uses for the text operator include

- imbedding a numeric value within a text string
- using it in an action verification expression
- using it in an alarm message
- sending out to a remote PLC display panel

The expression `“Flow rate is” & TEXT(Flow, “0.00”)& “gpm”` produces a text signal of `Flow rate is 141.23 gpm`, assuming `Flow` is a numeric signal whose value rounds to 141.23.

Comparison Operators

Comparison operators compare two numeric values and produce a logical result of either TRUE or FALSE (on or off).

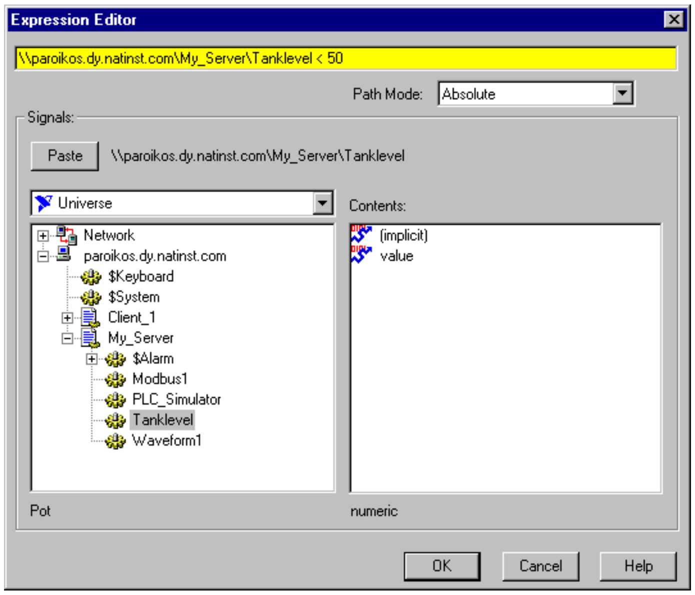
Table 1-3. Comparison Operators

<	Is less than
>	Is greater than
<=	Is less than or equal to
>=	Is greater than or equal to

Table 1-3. Comparison Operators (Continued)

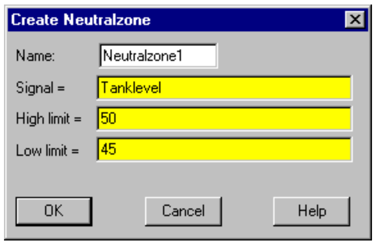
=	Is equal to
<>	Is not equal to

When setting up process control strategies, you often want to compare two numeric values. For example, you can use the result of the following expression to control a tank fill valve: `TankLevel<50` is TRUE while `TankLevel` is less than 50 and FALSE while `TankLevel` is greater than or equal to 50. The use of the < operator makes this a logical test, returning true or false depending on how the expression evaluates.



You can then connect this logical result to a PLC or RTU to control the fill valve so that the valve opens when the tank level drops below 50 and closes when the tank level rises above 50. Notice, however, that this method can cause the valve to fluctuate between open and closed too often if the tank level hovers around 50.

Neutralzone is an object class with built-in dead band that is more appropriate for controlling the tank fill valve. The following dialog box shows a better way to control a valve than the expression in the previous figure.


Create Neutralzone

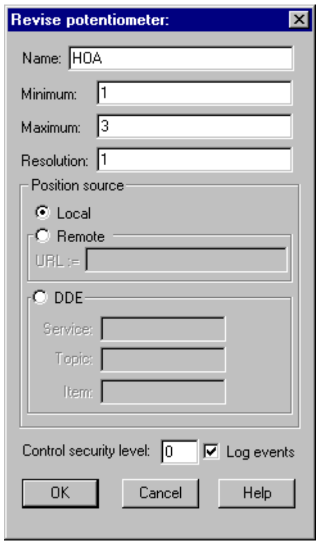
Name: Neutralzone1

Signal = Tanklevel

High limit = 50

Low limit = 45

Avoid using the *is equal to* (=) and *is not equal to* (<>) comparison operators to compare an analog value from a PLC or the result of mathematical calculations in an expression. Because numeric (floating point) values have about 17 significant digits, `TankLevel = 40` might never be exactly true. If you know the signals are integer values, such as the numeric signal from the potentiometer in the following dialog box, use the = and <> operators.


Revise potentiometer:

Name: HOA

Minimum: 1

Maximum: 3

Resolution: 1

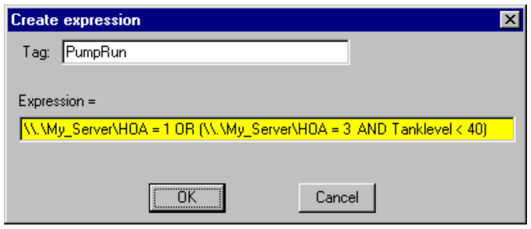
Position source: [✓ Local] Remote DDE

Control security level: 0

Log events ✓

In the following **Create expression** dialog box, a three-position switch is created with the Pot object class, where 1=Hand, 2=Off, and 3=Auto.

Next, an expression object is created that turns the pump on in two conditions: if the `HOA` switch is in the Hand position, or if the switch is in the Auto position and the `TankLevel` is less than 40.



Tag: PumpRun

Expression = \\.\My_Server\HOA = 1 OR (\\.\My_Server\HOA = 3 AND Tanklevel < 40)

Notice that the HOA signal is exactly 1, 2, or 3.

Tip You could also use the Lookout*Direct* Radiobutton object instead of creating an HOA Pot.

Expression Functions

There are over 50 built-in expression functions, generally classified as follows:

- logical functions
- lookup functions
- mathematical functions
- statistical functions
- text functions
- trigonometric functions
- date/time functions
- quality functions

The remainder of this chapter describes each function, specifies the syntax, and provides an example on how to use the function.

Logical Functions

AND	Syntax	*`logical1`* `AND` *`logical2`* `AND` *`logical3`*`...`
	Example	`Switch1 AND Pot1 > 50.0`
	Description	If `Switch1` is on *and* `Pot1` is greater than 50.0, return TRUE, otherwise return FALSE. Returns TRUE if *all* logical parameters are TRUE.

FALSE	Syntax	`FALSE`
	Example	`False`
	Description	Returns the logical value FALSE. Accepts no parameters.
IF	Syntax	`IF (`*`logical`*`, `*`result1`*`, `*`result2`*`)`
	Example	`IF(Switch1, 99.9, Pot1)`
	Description	If *`logical`* is TRUE, return *`result1`*, otherwise return *`result2`*. In the example, if `Switch1` is on, it outputs the value `99.9`. Otherwise, it outputs the value of `Pot1`.
LIF	Syntax	`LIF (`*`logical`*`, `*`logical result1`*`, `*`logical result2`*`)`
	Example	`LIF(Switch1, False, Pot1 > 50.0)`
	Description	Included for compatibility with early versions of Lookout*Direct*. If *`logical`* is TRUE, return *`logical result1`*, otherwise return *`logical result2`*. In the example, if `Switch1` is on, it outputs the logical value `False`. Otherwise, it returns the logical result of `Pot1 > 50.0`.
NIF	Syntax	`NIF (`*`logical`*`, `*`numeric result1`*`, `*`numeric result2`*`)`
	Example	`NIF(Switch1, 99.9, Pot1)`
	Description	Included for compatibility with early versions of Lookout*Direct*. If *`logical`* is TRUE, return *`numeric result1`*, otherwise return *`numeric result2`*. In the example, if `Switch1` is on, it outputs the numeric value `99.9`. Otherwise it outputs the value of `Pot1`.
NOT	Syntax	`NOT (`*`logical`*`)` or `!`*`logical`*
	Example	`NOT(`*`Switch1`*`)` or `!`*`Switch1`*
	Description	If *`logical`* is TRUE, return FALSE, else return TRUE. In the example, if `Switch1` is on, return FALSE, but if `Switch1` is off, return TRUE.
OR	Syntax	*`logical1`*` OR `*`logical2`*` OR `*`logical3`*`...`
	Example	`Switch1 OR Pot1 > 50.0`
	Description	Returns TRUE if at least one logical parameter is TRUE. In the example, if either `Switch1` is on or `Pot1` is greater than 50.0, or both, the expression returns TRUE. If neither condition is true, it returns FALSE.

TIF	Syntax	`TIF (`*`logical`*`, `*`text result1`*`, `*`text result2`*`)`
	Example	`TIF(Switch1, “Text Message”, TextInput1)`
	Description	Included for compatibility with early versions of Lookout*Direct*. If *`logical`* is TRUE, return *`text result1`*, otherwise return *`text result2`*. Notice the use of quotation marks. In the example, if `Switch1` is on, it outputs `Text Message`; otherwise, it outputs the current text value of `TextInput1`.
TRUE	Syntax	`TRUE`
	Example	`TRUE`
	Description	Returns the logical value TRUE. Accepts no parameters.
XOR	Syntax	*`logical1`* `XOR` *`logical2`* `XOR` *`logical3`*`...`
	Example	`Switch1 XOR Pot1 > 50.0`
	Description	Returns TRUE *if only one* logical parameter is TRUE. In the example, if `Pot1` is greater than 50 and `Switch1` is on, it outputs FALSE because both logical parameters are TRUE. If `Pot1` is less than 50 and `Switch1` is on, it outputs TRUE because only one logical parameter is TRUE.

Lookup Functions

CHOOSE	Syntax	`CHOOSE (`*`numeric`*`, `*`value1`*`, `*`value2`*`, `*`value3`*`,...)`
	Example	`CHOOSE(Pot1, “Switch1”, “TRUE”, “Pb1”)`
	Description	Returns the value parameter corresponding to the integer portion of the `numeric` parameter. If the *`numeric`* parameter is less than 2.0, `CHOOSE` returns *`value1`*. If the *`numeric`* parameter is greater than the number of value parameters, `CHOOSE` returns the last logical parameter listed. In the example, if the value of `Pot1` is .5, 1, or 1.4, it returns the value of `Switch1`. If `Pot1` is 2.0 or 2.8, `CHOOSE` returns the logical value `TRUE`. If `Pot1` is 3.0 or higher, `CHOOSE` returns the value of `Pb1`.

LCHOOSE	Syntax	`LCHOOSE (`*`numeric`*`, `*`logical1`*`, `*`logical2`*`, `*`logical3`*`,...)`
	Example	`LCHOOSE(Pot1, Switch1, TRUE, Pb1)`
	Description	Included for compatibility with early versions of Lookout*Direct*. Returns the logical parameter corresponding to the integer portion of the `numeric` parameter. If the *`numeric`* parameter is less than 2.0, `LCHOOSE` returns *`logical1`*. If the `numeric` parameter is greater than the number of logical parameters, `LCHOOSE` returns the last logical parameter listed.
NCHOOSE	Syntax	`NCHOOSE (`*`numeric`*`, `*`numeric1`*`, `*`numeric2`*`, `*`numeric3`*`,...)`
	Example	`NCHOOSE(Pot1, Pot2, Pot3, 14.3)`
	Description	Included for compatibility with early versions of Lookout*Direct*. Returns the *`numericx`* parameter corresponding to the integer portion of the `numeric` parameter. If the *`numeric`* parameter is less than 2.0, `NCHOOSE` returns *`numeric1`*. If the *`numeric`* parameter is greater than the number of *`numericx`* parameters, `NCHOOSE` returns the last *`numericx`* parameter listed.
TCHOOSE	Syntax	`TCHOOSE (`*`numeric`*`, `*`text1`*`, `*`text2`*`, `*`text3`*`,...)`
	Example	`TCHOOSE(Pot1, “Auto”, “Manual”, “Local”, “Locked”)`
	Description	Included for compatibility with early versions of Lookout*Direct*. Returns the text parameter corresponding to the integer portion of the *`numeric`* parameter. If the *`numeric`* parameter is less than 2.0, `TCHOOSE` returns *`text1`*. If the *`numeric`* parameter is greater than the number of text parameters, `TCHOOSE` returns the last logical parameter listed.

Mathematical Functions

ABS	Syntax	`ABS(`*`numeric`*`)`
	Example	`ABS(Pot1 – 50.0)`
	Description	Returns the absolute value of *`numeric`*. In the example, if `Pot1` is zero, the function returns `50.0`.

EXP	Syntax	`EXP(`*`numeric`*`)`
	Example	`EXP(Pot1)`
	Description	Returns the base of the natural logarithm, *e* (2.71828182), raised to the power of *`numeric`*. In the example, if `Pot1` is 2.0, the function returns 2.71828182^2 or approximately 7.38906. EXP is the inverse of the function, LN. To calculate the values of other powers, use the exponentiation operator (^).
FACT	Syntax	`FACT(`*`numeric`*`)`
	Example	`FACT(4.2)`
	Description	Returns the factorial of the integer portion of *`numeric`*. In the example, the function returns the factorial of 4, or 1*2*3*4, or 24. The factorial of zero, FACT(0), or any number less than zero is one (1).
INT	Syntax	`INT(`*`numeric`*`)`
	Example	`INT(4.2)`
	Description	Rounds *`numeric`* down to the nearest integer. In the example, the `INT` function returns 4. Notice also that `INT(–8.5) = –9`. See also *TRUNC*.
LN	Syntax	`LN(`*`numeric`*`)`
	Example	`LN(PLC.AI1)`
	Description	Returns the natural logarithm of *`numeric`*. In the example, if `PLC.AI1` = 1000.0, the function returns 6.90776.
LOG	Syntax	`LOG(`*`numeric1, numeric2`*`)`
	Example	`LOG(Pot1,2)`
	Description	Returns the logarithm of *`numeric1`* to the *`numeric2`* base, where *`numeric1`* is a positive real number. In the example, if `Pot1` = 8.0, the function returns 3.0.
LOG10	Syntax	`LOG10(`*`numeric`*`)`
	Example	`LOG10(Pot1)`
	Description	Returns the base-10 logarithm of *`numeric`*, where *`numeric`* is a positive real number. In the example, if `Pot1` = 100.0, the function returns 2.0.

MOD	Syntax	`MOD(`*`numeric1`*`, `*`numeric2`*`)`
	Example	`MOD(50,8)`
	Description	Returns the modulus (remainder) of *`numeric1`* divided by *`numeric2`*, where *`numeric2`* is not equal to zero. In the example, 50 divided by 8 equals 6 with a remainder of 2. Therefore, the function returns 2.
PI	Syntax	`PI()`
	Example	`PI()`
	Description	Returns an approximation of PI: 3.1415927. Accepts no parameters.
PRODUCT	Syntax	`PRODUCT(`*`numeric1`*`, `*`numeric2`*`, `*`numeric3`*`,...)`
	Example	`PRODUCT(2,4,6,10)`
	Description	Returns the product of all the numerics. In the example, the function returns 2*4*6*10 or 480.
RAND	Syntax	`RAND()`
	Example	`RAND() * Pot1`
	Description	Generates a new random number between zero and one every time the formula that this function is a part of is recalculated. Accepts no parameters. In the example, every time the value of `Pot1` changes, the function generates a new random number and multiplies it by the value of `Pot1`.
ROUND	Syntax	`ROUND(`*`numeric1`*`, `*`numeric2`*`)`
	Example	`ROUND(Pot1,2)`
	Description	Rounds the value of *`numeric1`* to *`numeric2`* decimal places. In the example, if `Pot1` equals 15.745, the function returns 15.75. If *`numeric2`* equals zero, the function returns an integer. If *`numeric2`* is less than zero, the function returns zero.
SIGN	Syntax	`SIGN(`*`numeric`*`)`
	Example	`SIGN(Pot1)`
	Description	Returns 1 if *`numeric`* is positive, 0 if *`numeric`* is 0, –1 if *`numeric`* is negative. In the example, if `Pot1` is –0.00001, the function returns –1.

SQRT	Syntax	`SQRT(`*`numeric`*`)`
	Example	`SQRT(ABS(Pot1))`
	Description	Returns the square root of *`numeric`*, where *`numeric`* is a positive real number. In the example, if `Pot1` is –25.0, the absolute function first converts the Pot value to positive 25 and the square root function calculates $\sqrt{25}$ = 5.
TRUNC	Syntax	`TRUNC(`*`numeric`*`)`
	Example	`TRUNC(8.9)`
	Description	Truncates *`numeric`* to its integer component by removing its fractional part. In the example, the `TRUNC` function returns 8. Notice also that `TRUNC(–8.9) = –8`. See also *INT*.

Statistical Functions

AVG	Syntax	`AVG(`*`numeric1, numeric2, numeric3`*`,...)`
	Example	`AVG(2,4,–6,12)`
	Description	Returns the arithmetic mean (average) of all the numerics listed. This function requires at least two numeric parameters. In the example, the function returns (2+4–6+12) / 4 or 3.0.
MAX	Syntax	`MAX(`*`numeric1, numeric2, numeric3`*`,...)`
	Example	`MAX(2,4,–6,12)`
	Description	Returns the highest value of all the numeric values listed. This function requires at least two numeric values. In the example, the function returns 12.
MIN	Syntax	`MIN(`*`numeric1, numeric2, numeric3`*`,...)`
	Example	`MIN(2,4,–6,12)`
	Description	Returns the lowest value of all the numeric values listed. This function requires at least two numeric values. In the example, the function returns –6.

STDEV	Syntax	`STDEV(`*`numeric1`*`, `*`numeric2`*`, `*`numeric3`*`,...)`
	Example	`STDEV(2.9,4.5,5.0,4.3,3.8)`
	Description	Returns the sample standard deviation of the numeric values listed. (Standard deviation is a measure of dispersion, calculated as the positive square root of the variance.) This function calculates the standard deviation using the non-biased or *n*–1 method. This function requires at least two numeric values. In the example, the function returns 0.796869.
STDEVP	Syntax	`STDEVP(`*`numeric1`*`, `*`numeric2`*`, `*`numeric3`*`,...)`
	Example	`STDEVP(2.9,4.5,5.0,4.3,3.8)`
	Description	Returns the standard deviation of a full population of numeric values. This function calculates the standard deviation using the biased or *n* method. This function requires at least two numeric values. In the example, the function returns 0.712741.
SUM	Syntax	`SUM(`*`numeric1`*`, `*`numeric2`*`, `*`numeric3`*`,...)`
	Example	`SUM(2,4,–6,12)`
	Description	Returns the sum of all the numeric values listed. This function requires at least two numeric values. In the example, the function returns 12.
VAR	Syntax	`VAR(`*`numeric1`*`, `*`numeric2`*`, `*`numeric3`*`,...)`
	Example	`VAR(2.9,4.5,5.0,4.3,3.8)`
	Description	Returns the sample variance of the numeric values listed. (Variance is a measure of dispersion.) This function calculates the variance using the non-biased or *n*–1 method. This function requires at least two numeric values. In the example, the function returns 0.635.
VARP	Syntax	`VARP(`*`numeric1`*`, `*`numeric2`*`, `*`numeric3`*`,...)`
	Example	`VARP(2.9,4.5,5.0,4.3,3.8)`
	Description	Returns the population variance of the numeric values listed. This function calculates the variance using the biased or *n* method. This function requires at least two numeric values. In the example, the function returns 0.508.

Text Functions

EXACT	Syntax	`EXACT(`*`text1`*`, `*`text2`*`)`
	Example	`EXACT(`*`TextEntry1`*`, “batch a”)`
	Description	Returns TRUE if *`text1`* exactly matches *`text2`*, otherwise returns FALSE. This function is case sensitive. In the example, the function returns TRUE if the text result of `TextEntry1` exactly matches `batch a` (notice that the entry in this example is all lowercase).
FIND	Syntax	`FIND(`*`text1`*`, `*`text2`*`, `*`numeric`*`)`
	Example	`FIND(“ON”, ”Reactor A is ON”, 1)`
	Description	Searches for *`text1`* within *`text2`* starting after *`numeric`* characters, and returns the position of the first character where the match begins. This function is case sensitive. The output of this function is numeric. It returns 0 if no match is found. In the example, the function searches the entire string for the word, `ON` and returns 14, indicating the position of the first character of the word. Also see *SEARCH*.
FIXED	Syntax	`FIXED(`*`numeric1`*`, `*`numeric2`*`)`
	Example	`“The flow is” & FIXED(Pot1,2)`
	Description	Rounds the value of *`numeric1`* to *`numeric2`* decimal places and then converts *`numeric1`* to a text string. In the example, if the value of `Pot1` equals `123.456`, the `FIXED` function returns the text value `123.46`. Thus, the entire text string would read `The flow is 123.46`. If *`numeric2`* is negative or omitted, *`numeric1`* is rounded to the nearest whole number. For example, `FIXED(123.588)` returns `124`. See also *TEXT*.
LEFT	Syntax	`LEFT(`*`text`*`, `*`numeric`*`)`
	Example	`LEFT(“Reactor A is ON”, 9)`
	Description	Retrieves the specified number of characters from the lefthand end of *`text`* and outputs it as a text value. In the example, the function returns `Reactor A`.

LEN	Syntax	`LEN(`*`text`*`)`
	Example	`LEN(“Reactor A is ON”)`
	Description	Returns the length of the text string (the number of characters in *`text`*). In the example, the function returns the numeric value 15.
LOWER	Syntax	`LOWER(`*`text`*`)`
	Example	`EXACT(LOWER(`*`TextEntry1`*`), “batch a”)`
	Description	Converts *`text`* to all lower case. In the example, the `LOWER` function ensures that a match is found whether `TextEntry1` content reads `Batch A`, `BATCH A`, or `batch A`.
MID	Syntax	`MID(`*`text`*`, `*`numeric1`*`, `*`numeric2`*`)`
	Example	`MID(“Reactor A is ON”,9,7)`
	Description	Retrieves *`numeric2`* characters from *`text`*, beginning at character *`numeric1`*. In the example, the function returns the text value `A is ON.`
PROPER	Syntax	`PROPER(`*`text`*`)`
	Example	`PROPER(“reactor A is ON”)`
	Description	Capitalizes the first character of each word in *`text`*. In the example, the function returns the text value `Reactor A Is On.`
REPLACE	Syntax	`REPLACE(`*`text1`*`, `*`numeric1`*`, `*`numeric2`*`, `*`text2`*`)`
	Example	`REPLACE(“Reactor A is ON”, 9, 1, “B”)`
	Description	Replaces *`numeric2`* characters with *`text2`* beginning with character *`numeric1`* in *`text1`*. In the example, the function returns the text value `Reactor B is ON.`
REPT	Syntax	`REPT(`*`text`*`, `*`numeric`*`)`
	Example	`REPT(“LookoutDirect”, Pot1)`
	Description	Repeats *`text numeric`* times. In the example, if the value of `Pot1` is 8, the function returns the text value, `LookoutDirect LookoutDirect LookoutDirect LookoutDirect LookoutDirect LookoutDirect LookoutDirect LookoutDirect.`

RIGHT	Syntax	`RIGHT(`*`text`*`, `*`numeric`*`)`
	Example	`RIGHT(“Reactor A is ON”, 7)`
	Description	Retrieves the specified number of characters from the righthand end of *`text`* and outputs it as a text value. In the example, the function returns the text value `A is ON`.
SEARCH	Syntax	`SEARCH(`*`text1`*`, `*`text2`*`, `*`numeric`*`)`
	Example	`SEARCH(“on”, ”Reactor A is ON”, 1)`
	Description	Finds *`text1`* within *`text2`* beginning at *`numeric`* character, and returns the position of the first character where the match begins. This function is not case sensitive. The output is numeric. It returns 0 if no match is found. In the example, the function searches the entire string for the word `ON, on, On,` or `oN` and returns 14, indicating the position of the first character of the word. Refer also to *FIND*.
TEXT	Syntax	`TEXT(`*`numeric`*`, `*`text`*`)`
	Example	`TEXT(Pot1, “0.00”)`
	Description	Included for compatibility with early versions of Lookout*Direct*. With polymorphic data, this function becomes unnecessary. Like `FIXED`, the `TEXT` function converts *`numeric`* to a textual value. While `FIXED` allows you to specify the number of decimal points, `TEXT` allows you specify a desired numeric format in the text parameter. If `Pot1` equals 12.3456, the function would return the textual value `12.35`. See *Numeric Formats* in Chapter 5, *Developer Tour*, in your *Getting Started with LookoutDirect* manual. See also *FIXED*.
TRIM	Syntax	`TRIM(`*`text`*`)`
	Example	`TRIM(“Reactor   A   is ON”)`
	Description	Trims multiple spaces from between words. In the example, there are multiple spaces on either side of the word, *A*. This function returns `Reactor A is ON`, eliminating all repeated spaces.

UPPER	Syntax	`UPPER(`*`text`*`)`
	Example	`UPPER(`*`text`*`)`
	Description	Converts *`text`* to all capital letters (upper case). In the example, the `UPPER` function ensures that a match is found whether `TextEntry1` content reads `Batch A`, `BATCH A`, or `batch A`.

Trigonometric Functions

ACOS	Syntax	`ACOS(`*`numeric`*`)`
	Example	`ACOS(Pot1)`
	Description	Returns the arccosine of *`numeric`* (that is, it returns the angle of the cosine you specify as *`numeric`*). *`Numeric`* must range between –1 and 1. The result is output in radians and ranges from 0 to π. In the example, if `Pot1` = 0.5, the function returns 1.0472 radians (60 degrees). You can express the arccosine in degrees by multiplying the result by 180/π.
ASIN	Syntax	`ASIN(`*`numeric`*`)`
	Example	`ASIN(Pot1)`
	Description	Returns the arcsine of *`numeric`* (that is, it returns the angle of the sine you specify as *`numeric`*). *`numeric`* is the sine of the angle and must range between –1 and 1. The resulting value is given in radians and ranges from –π/2 to π/2. In the example, if `Pot1` = –1, the function returns –1.5708.
ATAN	Syntax	`ATAN(`*`numeric`*`)`
	Example	`ATAN(Pot1)`
	Description	Returns the arctangent of *`numeric`* (that is, it returns the angle of the tangent that you specify as *`numeric`*). The resulting value is given in radians and ranges from –π/2 to π/2. In the example, if `Pot1` = 180, the function returns 1.56524 radians (about 90 degrees).

ATAN2	Syntax	`ATAN2(`*`numericX`*`, `*`numericY`*`)`
	Example	`ATAN2(5, 5)`
	Description	Returns the arctangent of the specified x- and y-coordinates (that is, it returns the angle of a line extending from the origin (0,0) to a point specified by the *`numericX`*, *`numericY`* coordinate pair that you specify). The resulting value is given in radians and ranges from greater than –π to π. In the example, the function returns 0.785398 radians (45 degrees).
COS	Syntax	`COS(`*`numeric`*`)`
	Example	`COS(Pot1)`
	Description	Returns the cosine of *`numeric`* where *`numeric`* is the angle in radians. In the example, if `Pot1` = 1.047, the function returns 0.500171. You convert degrees to radians by multiplying by π/180.
SIN	Syntax	`SIN(`*`numeric`*`)`
	Example	`SIN(Pot1)`
	Description	Returns the sine of *`numeric`* where *`numeric`* is the angle in radians. In the example, if `Pot1` = 3.14159, the function returns 1.2246E–16 (effectively zero). You convert degrees to radians by multiplying by π/180.
TAN	Syntax	`TAN(`*`numeric`*`)`
	Example	`TAN(Pot1*PI()/180)`
	Description	Returns the tangent of *`numeric`* where *`numeric`* is the angle in radians. In the example, the value of `Pot1` is 45, but it is in degrees, not radians. Convert it to radians by multiplying it by π/180. In this example, the function returns 1.

Date/Time Functions

NOW	Syntax	`NOW(`*`logical`*`)`
	Example	`NOW($Keyboard.F1)`
	Description	Returns a numeric value representing the current system date and time when any signal within the parentheses changes. The result is a floating point number in which the integer represents the date and the fraction represents the time of day. In the example, when you press the function key <F1>, the date and time are output as a single number, like 34738.3. If you change the **Numeric Format** of the number to `mm/dd/yy hh:mm:ss`, the same number would be shown as `02/08/95 07:49:02`.
TODAY	Syntax	`TODAY(`*`logical`*`)`
	Example	`TODAY($Keyboard.F1)`
	Description	Returns a numeric value representing the current system date when any signal within the parentheses changes. The result is an integer that represents the number of days that have passed since Jan. 1, 1900. In the example, when you press the function key F1, the date is output as a single number, such as `34738`. If you change the **Numeric Format** of the number to `mm/dd/yy`, the same number would be shown as `02/08/95`.

Note If you want to display the current time only, subtract the `TODAY` function from the `NOW` function.

Example: `NOW($Keyboard.F1) – TODAY($Keyboard.F1)`

In this example, if you want the result to update itself every second, replace `$Keyboard.F1` with a one second pulse timer.

Quality Functions

Lookout*Direct* monitors data quality and informs you if the data quality has gone bad. When you display data on a control panel directly through an expression, or when an object with a displayable element is connected to some other object and receives bad data, the quality problem is indicated by a red X that Lookout*Direct* superimposes over the display.

Some data members might not be displayed or may not connect to an object on some particular panel, but you may still need to monitor the quality of that data. You can use the `qgood` and `qbad` functions for a quick check of quality.

Also, although a red X tells you there is a data quality problem, you might need more specific information about what has actually gone wrong with the data. You can use the `qtext` function to report the specifics of what has gone wrong.

qtext	Syntax	`qtext(`*`expression,`* `[`*`delimiter`*`])`
	Example	`qtext(modbus.40001, "AND")`
	Description	Returns a text string specifying what elements of data quality have gone bad.
qgood	Syntax	`qgood(`*`expression`*`)`
	Example	`qgood(modbus.40001)`
	Description	Returns TRUE while the expression is good.
qbad	Syntax	`qbad(`*`expression`*`)`
	Example	`qbad(modbus.40001)`
	Description	Returns TRUE while the expression is bad.