NovaTech®

Advanced Math & Logic for Orion

Software Manual

March 1, 2021
Document Revision A

NovaTech, LLC
13555 West 107th Street
Lenexa, KS 66215
Phone (913) 451-1880
www.novatechautomation.com
orion.support@novatechautomation.com
orion.sales@novatechautomation.com

Data classification: Public use

The NovaTech Configuration Director (NCD) software may be installed on multiple computers as needed under the following conditions:

- The computers must be owned by the end user or its subsidiary.
- The NCD installation must be used for an Orion-related project.

All files installed by NCD are protected by copyright and may not be shared with any third party. By installing NCD on a computer, you agree to these terms and conditions.

Disclaimer

This manual contains information that is correct to the best of NovaTech's knowledge. It is intended to be a guide and should be used as such. It should not be considered a sole source of technical instruction, replacing good technical judgment, since all possible situations cannot be anticipated. If there is any question as to the installation, configuration, or use of this product, contact NovaTech, LLC at (913) 451-1880.

To ensure that the equipment described in this User's Manual, as well as all equipment connected to and used with it, operates in a satisfactory and safe manner, all applicable local and national codes that apply to installing and operating the equipment must be followed. Since these codes can vary geographically and can change with time, it is the user's responsibility to determine which codes and standards apply, and to comply with them.

Failure to follow the instructions provided in this manual, and/or failure to comply with applicable codes and safety standards can result in damage to this equipment, damage to connected devices, and/or serious injury to personnel.

All links to external websites have been verified as correct and appropriate at the time of the publication of this document. However, these links and websites, being outside of NovaTech LLC's control, are subject to change and may no longer be correct. In this case, please contact:

orion.support@novatechautomation.com

The documentation for the Orion products is structured as follows.

Manual name (see cover page of each manual)	Purpose
Quick Startup Guide	Describes out-of-the-box setup for quick installation.
User manuals: ▪ OrionLX/OrionLX+ ▪ OrionLXm ▪ Orion I/O ▪ OrionMX	▪ Description of Orion hardware and hardware options. ▪ List of software protocol options. ▪ NovaTech Configuration Director (NCD) software description: ○ Installation of software on PC ○ Information on running NCD ▪ User interface information for monitoring and maintenance: ○ Using the Orion Webpage ○ Using MMI console • Setup and operation of the video option (-MMB for OrionLX. MMC for OrionLX+)
Orion Applications Manual	Specific setup and operation of Orion features, such as firewall, SFTP access, installation of SSL certificate, access to Orion's SQL database with Microsoft Access® and Excel®, and others.
Protocol/Software Manuals	Software manuals explain all aspects of setup and operation of protocols such as DNP3 client and software options (Archive, Logic, DA Logic, etc.)
Tech Notes	Tech Notes provide solutions for general integration, such as scaling or setup of RS-485 networks.
Field Instructions	Field Instructions provide step-by-step instructions for installation of new hardware or software in the field.

See also Appendix E – Additional References for a list and locations of the recommended reference manuals.

Styles and Symbols

In this document, fonts, text styles and symbols are used to distinguish standard text from keyboard input, program text, GUI messages, and hyperlinks as follows. Warnings and safety notices are indicated with ANSI symbols.

Displayed text or symbol	Description
This is normal text.	Standard text.
See Port Options	Hyperlink to text in same document.
www.novatechautomation.com	Hyperlink to website.
orion.support@novatechautomtion.com	Clicking this link starts email client on the PC.
See *OrionLX User Manual*	Document name.
`Minimum value`	Menu item or text displayed by software.
`Name of the data point`	Text to be entered in input field or window.
Save	GUI button to be clicked.
`if frequency < 60.0 then`	Program code.
<Enter>, <Ctrl>+<G>, <G>	Key to be pressed.
	This yellow triangle indicates a warning that must be observed by the users in order to avoid possible equipment damage or personal injury.
	This yellow triangle indicates an electrical hazard.
	Electrostatic sensitive device requires proper handling and grounding procedures to avoid equipment damage.
	DANGER indicates an imminently hazardous situation which, if not avoided, will result in death or serious injury.
	WARNING indicates a potentially hazardous situation which, if not avoided, could result in death or serious injury.
	CAUTION indicates a hazardous situation which, if not avoided, may result in minor or moderate injury.

Note that depending on the Windows® display settings on the computer running NCD, some of the screen shot details may appear differently than shown in this manual. The screen shots in this manual have been taken using Windows 10®.

Table of Contents

List of Figures

List of Tables

I. Introduction

The Advanced Orion Math & Logic module provides the capability to perform math or logic operations on any input or output points in the Orion. The result of the operation can be written to a logic (virtual) input point which can be mapped to a server port and polled by a client station. The result can also be written to a logic (virtual) output point which can then be sent to a server device to execute a control.

Typical applications include the following.

- ORing Comm Fail points from multiple devices into a single Logic Comm Fail.
- Primary/Secondary decisions based on the Comm Fail point for the Primary and Secondary devices.
- Totaling of numeric values (kWh, A, etc.) received from multiple devices.
- Controlling outputs in server devices based upon input values.

The logic within an Orion is event-driven and consists of one or more functions. Each function typically handles a specific task, for example “Initialization”, “Open Breaker”, “Hourly Total”, etc.

There are different types of events which trigger the execution of a logic function. Each function is set up to be triggered by a specific event. There can be multiple logic functions, each triggered by a different event.

- Load Event: The Load event (default is DefaultStartFunction) is triggered only once, i.e. when the Orion boots up.
- Timer Event: One or more timers can be set up to run at certain intervals, e.g. 100ms or 5 seconds. Every time the timer’s value is reached, an event is generated which triggers the execution of the associated function(s).
- DataChange Event: Each point can trigger an event based on data Refresh or data changing outside of user deadband.
    - Refresh: Any point can be set up to trigger a DataChange event when the point is written to the real-time database. Such an event would then trigger the execution of the associated function(s).
    - DeadBand: Any point can be set up to trigger an event when the point’s value changes outside of the deadband range from the previous value. Such an event would then trigger the execution of the associated function(s).

The logic is set up by using an editing window within NCD. Advanced Math & Logic uses the Lua programming language. NCD provides helpful command information and examples.

II. Setting Up Logic in NCD

Logic is part of the Orion configuration, and is set up in NCD as follows. After opening the desired .ncd file (`File`, `Open`) in the NCD main menu, navigate to the `Configure` menu, then to `Logic`. Select `Advanced Math & Logic` (Figure 1) from the dropdown list.

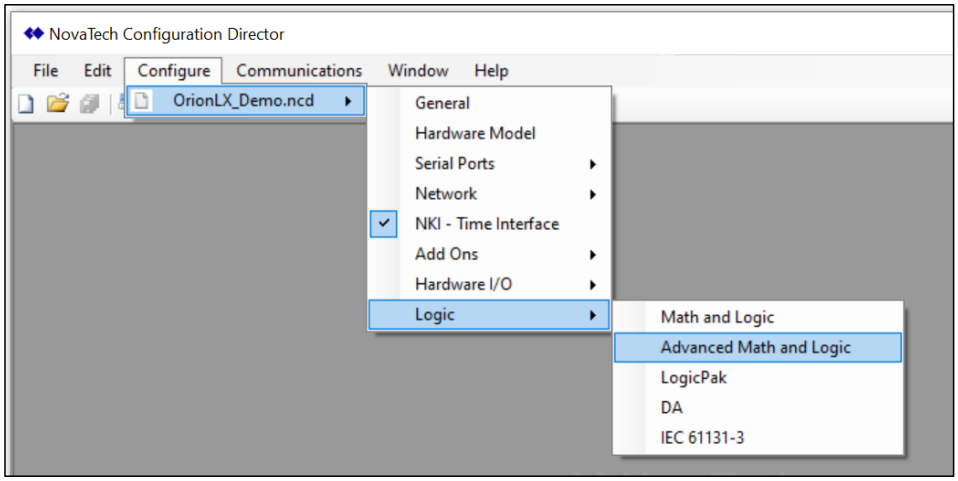


Figure 1: Logic Setup in NCD

III. Editing Logic in NCD

Introduction

Logic is set up and edited through the various elements on the `Standard` and `Functions` tabs as shown in the following window. This chapter will discuss all aspects and options. In the next chapter, several examples show how specific applications can be implemented using Advanced Math & Logic.

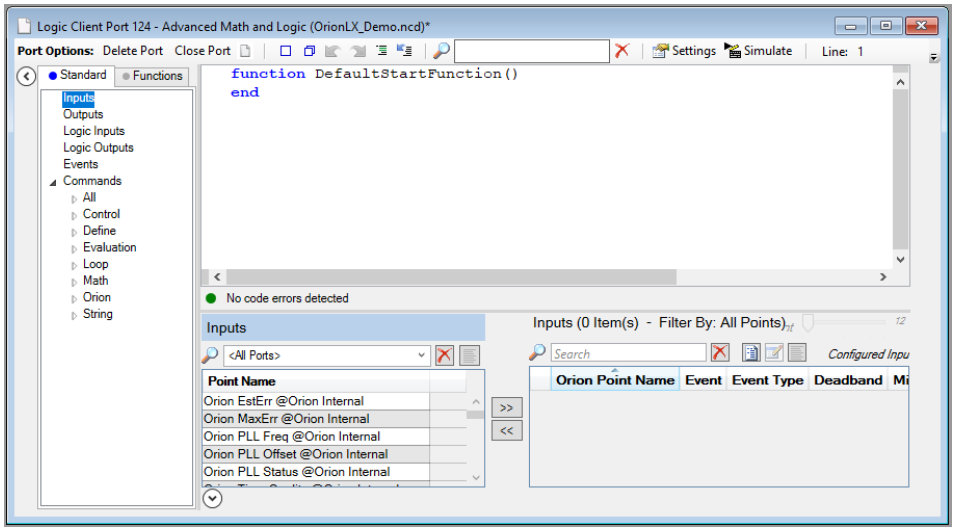


Figure 2: Main Logic Window

Port Options

In the upper left corner of each NCD display, the `Port Options` (Figure 3) menu is displayed.

Figure 3: Port Options

Command	Description
`Delete Port`	Removes the configuration, releasing the current logic. The logic is deleted and all associated points are deleted from the overall configuration.
`Close Port`	Closes the logic window to allow for the configuration of other ports. An * after the file name indicates that there are unsaved changes. Click on the  icon in the main NCD window to save the configuration before closing.

Table 1: Logic Options

Logic Editor

The NCD Logic Editor (Figure 4) is a standard Windows-style text editor. It supports standard text functions such as `Cut`, `Copy`, `Paste`, `Find`, and `Replace`. In addition, it provides color-coding for certain logic statements and symbols.

The text functions, as well as the color settings and find/replace options, are made available by right-clicking within the NCD `Logic` window. The color settings, fonts, etc. can be customized by selecting `Properties` (Figure 4).

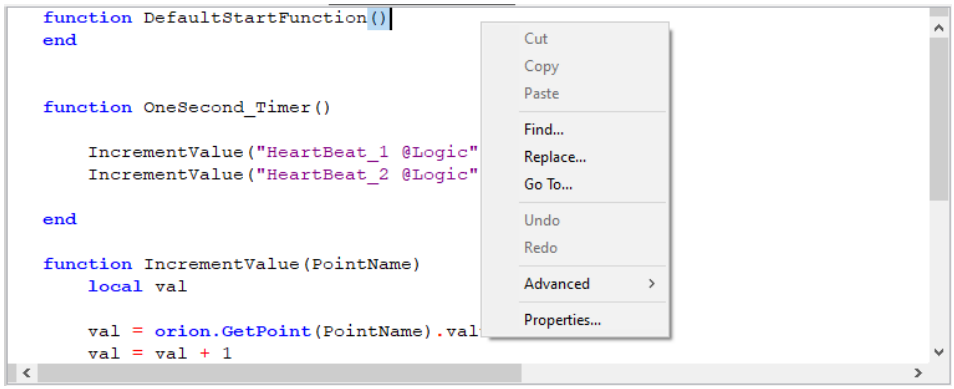


Figure 4: NCD Logic Editor

In the example shown (Figure 5), the `Keywords` are set to be displayed in Blue color, `Bold` font style, and `Courier New 11pt` font.

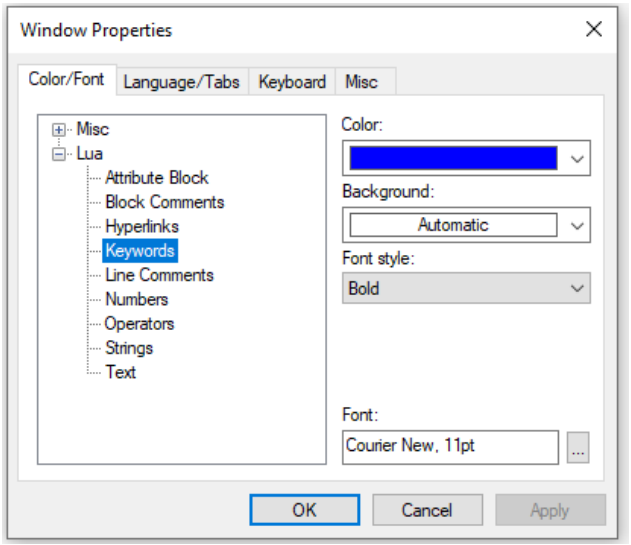


Figure 5: NCD Logic Editor Properties

Standard Tab

NCD provides the following Standard tab tree selections for logic setup. All other ports on the Orion should be configured prior to setting up Advanced Math & Logic so that the desired points are available for Advanced Math & Logic. The table below (Table 2) briefly describes each choice. Each selection is described in greater detail in paragraphs following the table below.

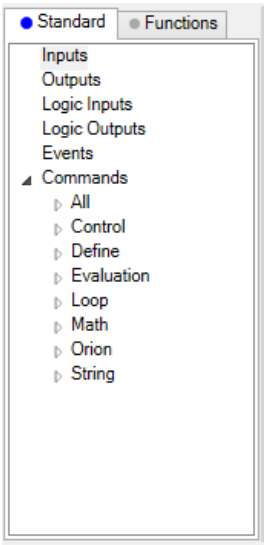


Figure 6: Standard Tab Options

Tab	Description
Inputs	Inputs are defined on Orion client ports. Once the Orion client ports have been configured, they will appear in the point list.
Outputs	Outputs are defined on Orion client ports. Once the Orion client ports have been configured, they will appear in the point list.
Logic Inputs	Logic input points can be mapped to Orion server ports (i.e. one or more client stations can poll the value).
Logic Outputs	Logic output points can be mapped to Orion client ports (i.e. the new value is sent to one or more server devices).
Events	The events specify when (on startup, timer-based, or change-based) the functions run. One or more events can be specified.
Commands	For each event, a specific function must be defined. The Commands selection provides syntax and examples for built in functions and logic operators. The functions/operators can be filtered by All, Control, Define, Evaluation, Loop, Math, Orion and String by selecting the appropriate tree element.

Table 2: Standard Tab Descriptions

Inputs Menu

Under the `Inputs` menu (Figure 7), the Orion input points which are to be monitored by logic must be set up. The input points are polled from server devices through the Orion's client ports (such as DNP client, Modbus client, SEL client). This means that the input points to be monitored in logic must first be created in a client protocol.

All available points are listed in the `Inputs List` box on the left.

Any input point to be used in logic must be highlighted by first selecting `Inputs` and then selecting the point in the `Point Name` list. Then move the point to the `Configured Inputs` box by clicking on the >> button. Likewise, points that are not needed in `Configured Inputs` can be moved out with <<.

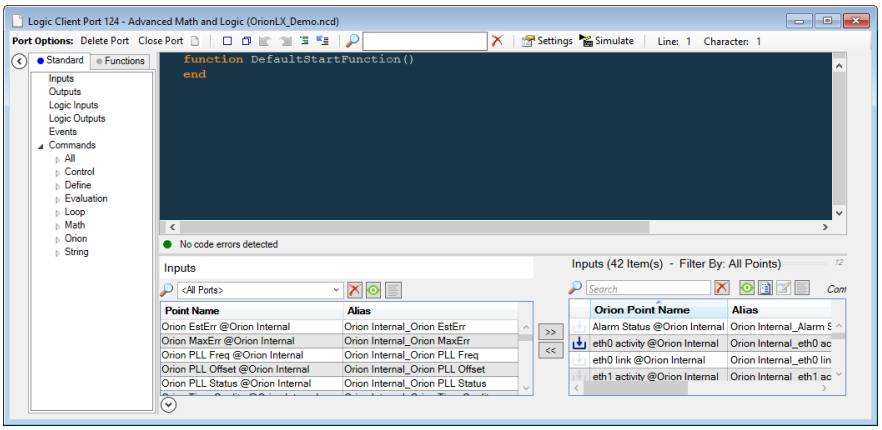


Figure 7: Inputs Menu

For a complete description of all available tools and buttons in this menu, see Appendix B – Tools and Buttons on the Input and Output Menus.

The following tables describe the input point parameters.

Parameter	Description
Event	All functions available in the `Event` list must first be set up under the `Events` menu as a `DataChange` event type. If an input point is linked to a `DataChange` function, it will call that function every time that input point is written to when configured as a `Refresh` event type, or when the value exceeds the deadband value when configured as a `Deadband` event type. Multiple input and/or output points can be assigned to the same event/function. This means that if any point(s) change, the corresponding function is executed. If an input point is configured with an event/function and is written to in a `Timer` event type, the `Event` field is generally left blank.
Event Type	Specifies when the `DataChange` event/function is executed. The value `Refresh` causes the function to be executed every time the input point is written to the real-time database. `Deadband` causes the function to be executed every time its real-time database value exceeds the deadband range. The `DataChange` function will be called when the input's online/offline status or string attribute change, regardless of the `Refresh` or `Deadband` event type.
Deadband	When the event type is `Deadband` and the current point value changes by more than the specified deadband value, an event is generated. Example: A deadband of 0.9 would detect a change of 0 to 1 (because 1 exceeds the deadband of 0.9), and trigger an event. In this example, if the point value changes from 0 to 0.5, no event would be generated because the value change is less than 0.9. Default: 10% of the point's min/max range.
Min Value **Max Value**	Each input point is set up with a minimum and maximum value. Points default to the min and max values defined on their client port. Internally, the Orion uses the current value of the point, and stores the normalized value, based on the minimum and maximum values[1]. Depending on the application, the minimum and maximum values might be desired to be different than the input point as used on the client port.
Queue Size	Specifies the number of events to be queued. When not 0, a per point queue holds events for retrieval using the orion.GetQueuedPoint command. orion.GetQueuedPoint allows logic to retrieve momentary changes, such as recloser operations.

Table 3: Inputs Menu Parameters

Button	Description
	Inserts the associated point name from the `Configured Inputs` box at the current cursor position in the `Logic` editor (Figure 7).

Table 4: Insert Button

[1] For a complete discussion of scaling, refer to the *Analog/Accumulator Scaling Technical Note*.

Outputs Menu

Under the `Outputs` menu (Figure 8), the Orion output points which are to be monitored by logic must be set up. The output points generally receive their settings from other client station(s) to which the Orion is a server.

All available points are listed in the `Outputs List` on the left.

Any output point to be used in logic must be highlighted by first selecting `Inputs` and then selecting the point in the `Point Name` list. Then move the point to the `Configured Inputs` box by clicking on the >> button. Likewise, points that are not needed in `Configured Outputs` can be moved out with << .

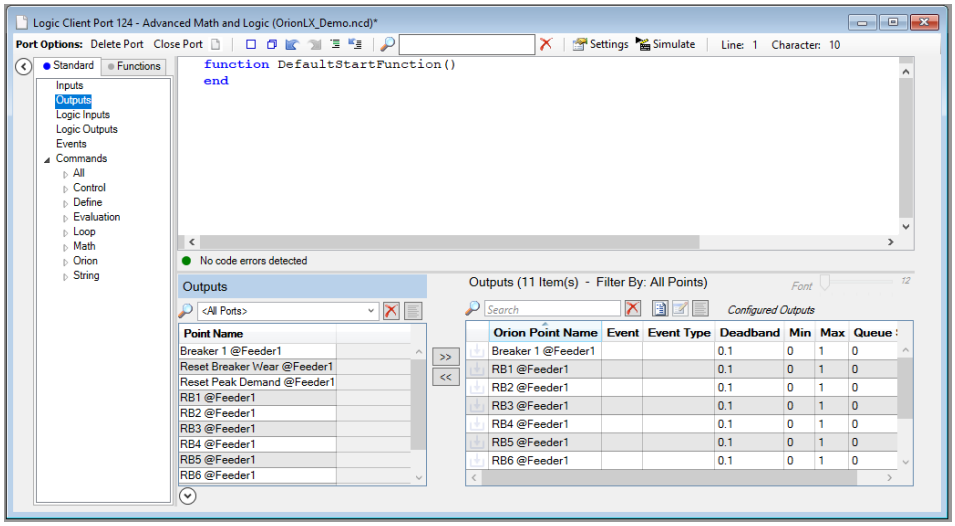


Figure 8: Outputs Menu

For a complete description of all available tools and buttons in this menu, see Appendix B – Tools and Buttons on the Input and Output Menus.

The following tables describe the output point parameters.

Parameter	Description
Event	All functions available in the `Event` list must first be set up under the `Events` menu with a `DataChange` event type. If an output point is linked to a `DataChange` function, it will call that function every time that output point is written to when configured as a `Refresh` event type, or when the value exceeds the deadband value when configured as a `Deadband` event type. Multiple input and/or output points can be assigned to the same event/function. This means that if any point(s) change, the corresponding function is executed. If an output point is configured with an event/function and is written to in a `Timer` event type, the `Event` field is generally left blank.
Event Type	Specifies when the `DataChange` event/function is executed. The value `Refresh` causes the function to be executed every time the output point is written to the real-time database. `Deadband` causes the function to be executed every time its real-time database value changes outside of the deadband range. The `DataChange` function will be called when the output's string attribute changes, regardless of the `Refresh` or `Deadband` event type.
Deadband	When the event type is `Deadband` and the current point value changes by more than the specified deadband value, an event is generated. Example: A deadband of 0.9 would detect a change of 0 to 1 (because 1 exceeds the deadband of 0.9), and trigger an event. In this example, if the point value changes from 0 to 0.5, no event would be generated because the value change is less than 0.9. Default: 10% of the point's min/max range.
Min Value **Max Value**	Each output point is set up with a minimum and maximum value. Points default to the min and max values defined on their client port. Internally, the Orion takes the current value of the point, and stores the normalized value, based on the minimum and maximum values[1]. However, depending on the application, the minimum and maximum values might be desired to be different than the output point as used on the client port.
Queue Size	Specifies the number of events to be queued. When not 0, a per point queue holds events for retrieval using the orion.GetQueuedPoint command. orion.GetQueuedPoint allows logic to retrieve momentary changes, such as recloser operations.

Table 5: Outputs Menu Parameters

Button	Description
	Inserts the currently highlighted point name in the `Configured Outputs` box at the current cursor position in the `Logic` editor (Figure 8).

Table 6: Outputs Menu Button

[1] For a complete discussion of scaling, refer to the *Analog/Accumulator Scaling Technical Note*.

Logic Inputs Menu

The `Logic Inputs` menu (Figure 9) allows new virtual input points to be created for use in Advanced Math & Logic. Points added under the `Logic Inputs` menu can be used for the following two purposes.

- Logic input points can be mapped to any server port in the Orion. When a function writes the result of a math or a logic operation to a logic input point, the value can be polled by a client station connected to the Orion.
- The logic input points provide the capability to store internal logic values that can be synchronized between redundant Orions.

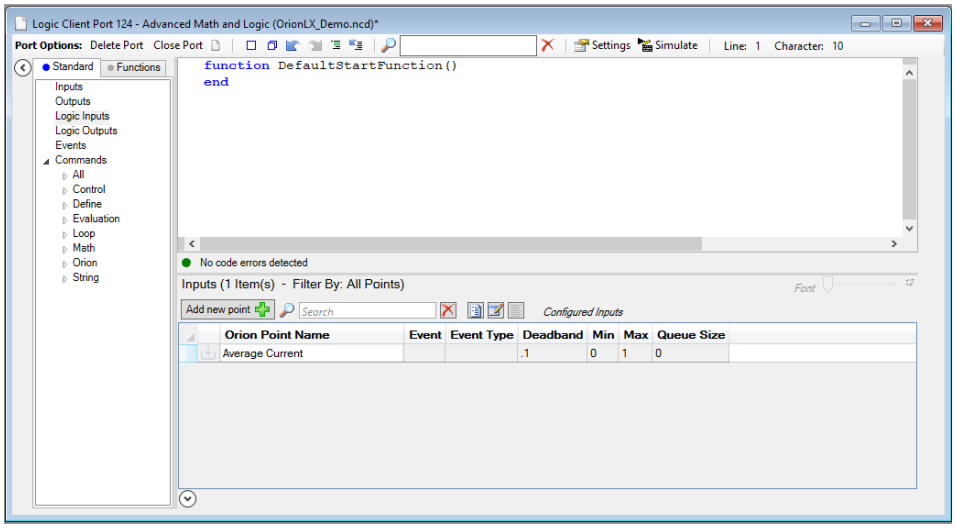


Figure 9: Logic Inputs Menu

For a complete description of all available tools and buttons in this menu, see Appendix B – Tools and Buttons on the Input and Output Menus.

The following tables describe the Logic Input point parameters.

Parameter	Description
Event	All functions available in the `Event` list must first be set up under the `Events` menu with `DataChange` event type. If a logic input point is linked to a `DataChange` function, it will call that function every time that input point is written to when configured as a `Refresh` event type, or when the value exceeds the deadband value when configured as a `Deadband` event type. Multiple input points and/or output points can be assigned to the same event/function. This means that if any point(s) change, the corresponding function is executed. If a logic input point is configured with an event/function and is written to in a `Timer` event type, the `Event` field is generally left blank.
Event Type	Specifies when the `DataChange` event/function is executed. The value `Refresh` causes the function to be executed every time the logic input point is written to the real-time database. `Deadband` causes the function to be executed every time its real-time database value changes outside of the deadband range. The `DataChange` function will be called when the input's online/offline status or string attribute change, regardless of the `Refresh` or `Deadband` event eype.
Deadband	If the current point value changes by more than the deadband value, an event is generated. Example: A deadband of 0.9 would detect a change of 0 to 1 (because 1 exceeds the deadband of 0.9), and trigger an event. In this example, if the point value changes from 0 to 0.5, no event would be generated because the value change is less than 0.9. Default: 10% of the point's min/max range.
Min Value **Max Value**	Each logic input point must be set up with a minimum and maximum value. The default values for discrete points are 0 and 1, and vary for analog points [1].
Queue Size	Specifies the number of events to be queued. When not 0, a per point queue holds events for retrieval using the orion.GetQueuedPoint command. orion.GetQueuedPoint allows logic to retrieve momentary changes, such as recloser operations.

Table 7: Logic Inputs Menu Parameters

Button	Description
	Inserts the currently highlighted point name in the `Configured Inputs` box at the current cursor position in the `Logic` editor (Figure 9).
	Creates a new row in the `Configured Inputs` box for the user to enter the logic input point name.

Table 8: Logic Inputs Menu Buttons

[1] For a complete discussion of scaling, refer to the *Analog/Accumulator Scaling Technical Note*.

Logic Outputs Menu

The `Logic Outputs` menu (Figure 10) allows new output points to be created for Advanced Math & Logic to write results to. Points added under this tab should be mapped to an Orion client port for the purpose of sending an output or control to an IED, PLC, etc. triggered from a logic function.

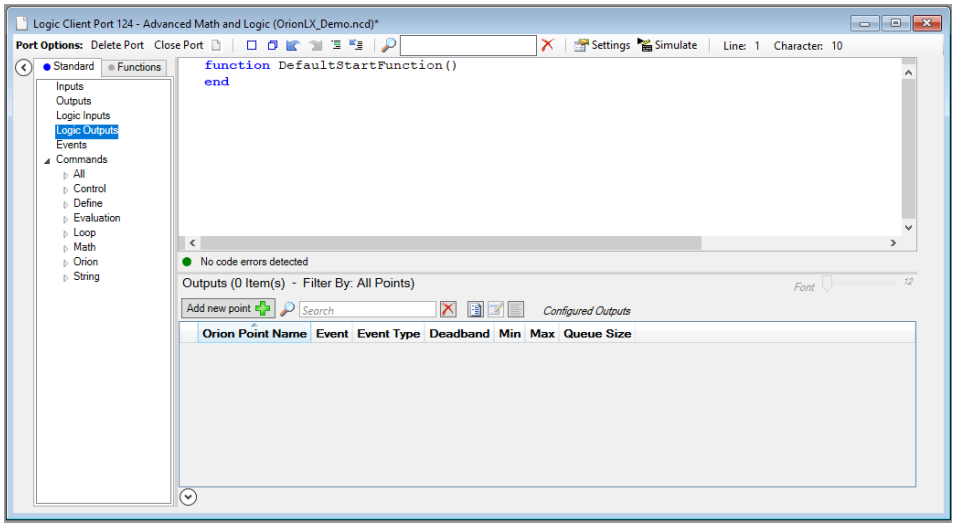


Figure 10: Logic Outputs Menu

For a complete description of all available tools and buttons in this menu, see Appendix B – Tools and Buttons on the Input and Output Menus.

The following tables describe the Logic Output point parameters.

Parameter	Description
Event	All functions available in the Event list must first be set up under the Events menu with a DataChange event type. If a logic output point is linked to a DataChange function, it will call that function every time that output point is written to when configured as a Refresh event type, or when the value exceeds the deadband value when configured as a Deadband event type. Multiple input and/or output points can be assigned to the same event/function. This means that if any point(s) change, the corresponding function is executed. If a logic output point is configured with an event/function and is written to in a Timer event type, the Event field is generally left blank.
Event Type	Specifies when the DataChange event/function is executed. The value Refresh causes the function to be executed every time the logic output point is written to the real-time database. Deadband causes the function to be executed every time its real-time database value changes outside of the deadband range. The DataChange function will be called when the string attribute changes, regardless of the Refresh or Deadband event type.
Deadband	If the current point value changes by more than the deadband value, an event is generated. Example: A deadband of 0.9 would detect a change of 0 to 1 (because 1 exceeds the deadband of 0.9), and trigger an event. In this example, if the point value changes from 0 to 0.5, no event would be generated because the value change is less than 0.9. Default: 10% of the point's min/max range.
Min Value **Max Value**	Each logic output point must be set up with a minimum and maximum value. The default values for discrete points are 0 and 1, and vary for analog points[1].
Queue Size	Specifies the number of events to be queued. When not 0, a per point queue holds events for retrieval using the orion.GetQueuedPoint command. orion.GetQueuedPoint allows logic to retrieve momentary changes, such as recloser operations.

Table 9: Logic Outputs Tab Parameters

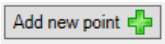
Button	Description
	Inserts the currently highlighted point name in the Configured Outputs box at the current cursor position in the Logic editor (Figure 9).
	Creates a new row in the Configured Outputs box for the user to enter the logic output point name.

Table 10: Logic Outputs Tab Buttons

[1] For a complete discussion of scaling, refer to the *Analog/Accumulator Scaling Technical Note*.

Events Menu

Under the `Events` menu, all events which will cause a function to be processed are defined. As new events are defined, the corresponding (still empty) functions are inserted into the logic editor. The functions must then be set up with the application-specific logic.

Functions triggered by the event type `Timer` will automatically have `_Timer` added to their name in the Logic editor.

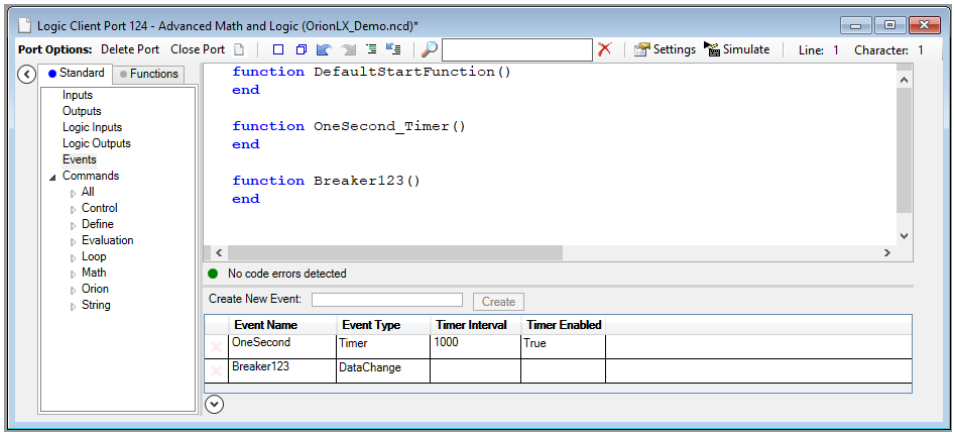


Figure 11: Events Menu

The following command buttons are available.

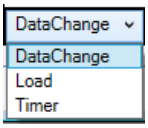
Button	Description
Create	Creates a new event. The `Create` button will be ebabled after a function name is entered in the `Create New Event` textbox. After clicking the `Create` button, a new row will be inserted with the `Event Name` entered. You can then change the `Event Type` to `DataChange`, `Load`, or `Timer`.  **Figure 12: Event Type Selection** For event types `DataChange` and `Load`, no additional parameters are required. For event type `Timer`, the `Timer Interval` and `Timer Enabled` parameters are required. `Timer Interval` specifies in milliseconds how often the associated function is executed. The minimum value is 0ms, and the maximum value is 99,999,999ms. `Timer Enabled` specifies whether the timer is enabled (`True` or `False`) upon startup of the Orion. Timers can also be enabled/disabled in logic.
	Deletes the event on that row. The associated function remains still in the `Logic` editor, but now has no event associated with it anymore.

Table 11: Events Menu Buttons

Data Change Events in Advanced Math & Logic

A function that is executed by a `DataChange` event can be associated to not just one but multiple data points. In this case, the function is executed if any one of these data points changes. The name of the data point that cause the execution can be obtained in the called function as follows.

```
function check_Values(strPointName)
    orion.PrintDiag("Execution triggered by the following point:")
    orion.PrintDiag(strPointName)
--other code to be executed
end
```

Commands Menu

The `Commands` menu lists common operators and functions. When one of these functions is highlighted (`for` is selected in the example below), the command parameters are listed in the lower right section with an example.

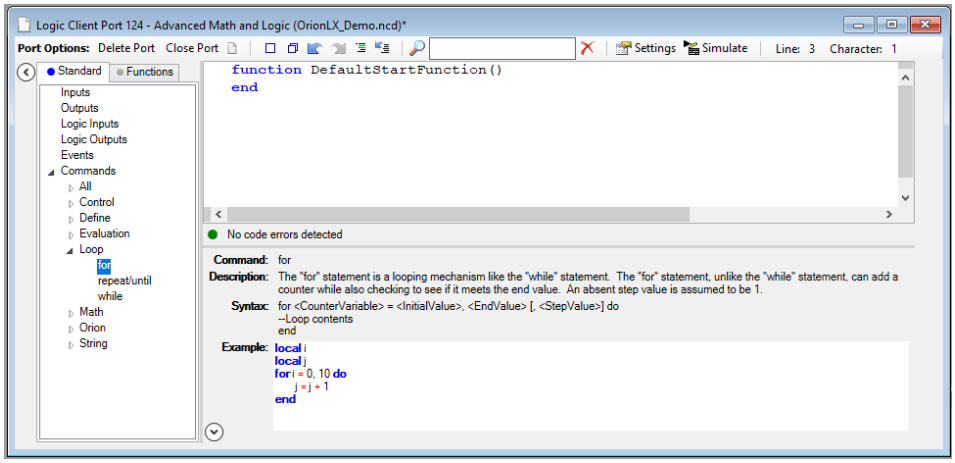


Figure 13: Commands Menu

Features	Description
`Insert Example`	Examples can be highlighted and either copied/pasted or dragged/dropped from the `Example` textbox to the `Logic` editor.
`Insert Command`	Operators and Functions can be inserted by dragging/dropping the command from the `Commands` menu to the `Logic` editor.

Table 12: Commands Menu Buttons

A complete list of commands, including a detailed description of all parameters, is included in Appendix A – List of Advanced Math & Logic Commands.

Functions Tab

The Functions tab provides a list of functions in the `Logic` editor in alphabetic order. Clicking on the function name will display that function in the `Logic` editor.

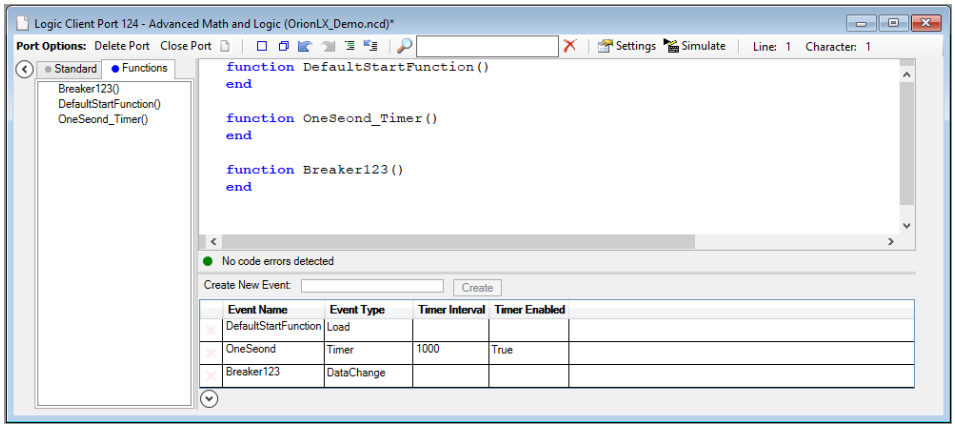


Figure 14: Functions Tab

Syntax Checking

NCD automatically checks syntax as code is entered in the Logic editor. The figure below shows an example with "No code errors detected" (syntax notification located between the `Logic` editor and the `Inputs` box).

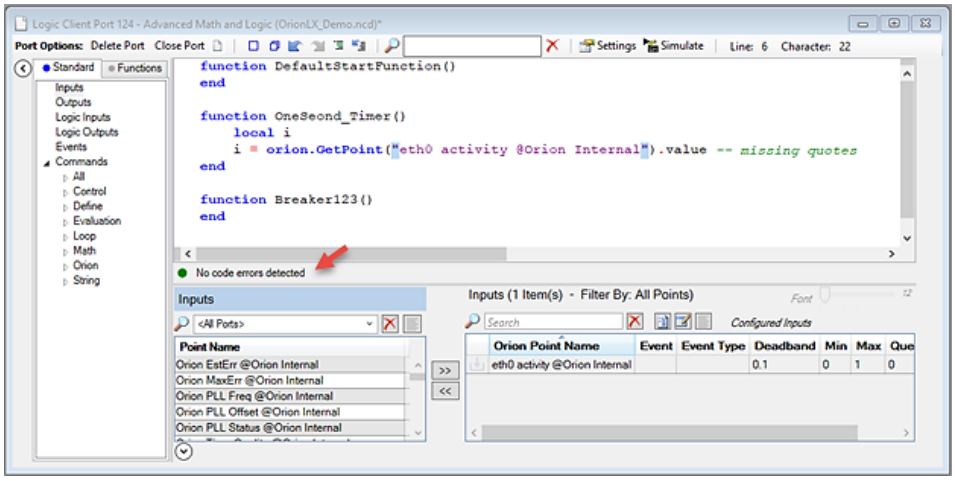


Figure 15: No Syntax Error

The figure below shows an example with an error because the point name was entered without quotes (syntax notification located between the `Logic` editor and the `Inputs` box).

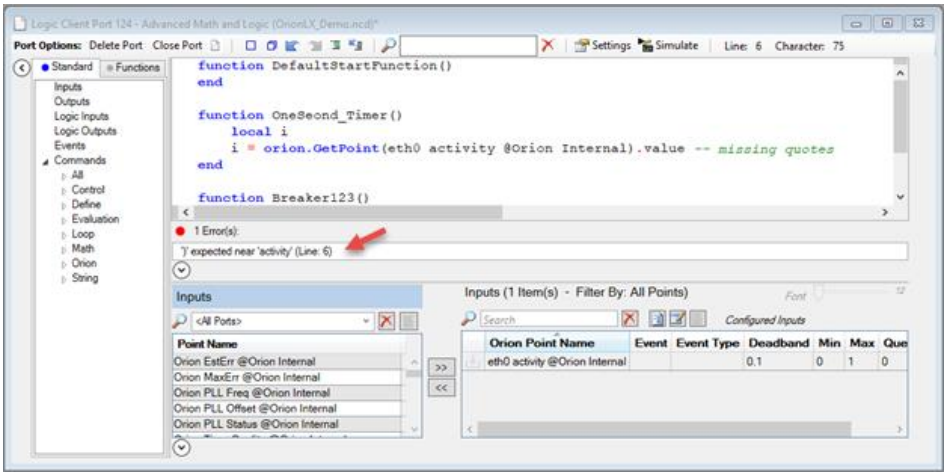


Figure 16: Syntax Error Detected

IV. Logic Editor Tools

NCD includes some common tools to aid in development. These tools allow you to maximize/restore the logic editor, undo/redo actions, comment/uncomment logic, search for text in the logic editor, and configure the logic editor. The following figure shows the Logic Editor Toolbar.

Figure 17: Logic Editor Toolbar

The following table describes each of the toolbar features.

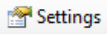
Feature	Tool Tip	Description
	Maximize code window	When this button is clicked, the left pane containing the tree and the bottom pane will be hidden to maximize the logic editor.
	Restore code window	When this button is clicked, the left pane containing the tree and the bottom pane will be restored.
	Undo	Undo the previous action. <Ctrl>+<Z> and <Alt>+<Backspace> are also supported.
	Redo	Redo the previous undo action. <Ctrl>+<Y> and <Ctrl>+<Shift>+<Z> are also supported.
	Comment out selection	Comment out the current line or selected lines.
	Uncomment selection	Uncomment the current selection.
	Search	Search for matching text in the logic editor.
	Settings	Set online status, show line numbers, and set font and colors as described below.

Table 13: Logic Editor Tools

Settings

The `Edit Settings` button opens the `Edit Settings` dialog box.

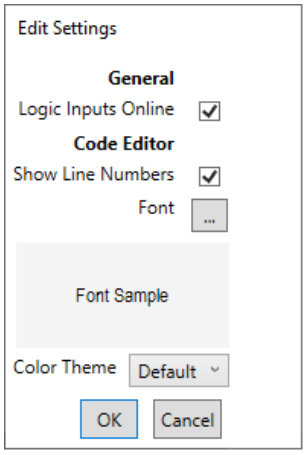


Figure 18: Edit Settings Dialog Box

Parameter	Description
`Logic Inputs Online`	When the `Logic Inputs Online` checkbox is checked, the Logic Input points will default to online status on startup. This setting defaults to checked.
`Show Line Numbers`	When the `Show Line Numbers` checkbox is checked, the editor will display line numbers on the left side of the logic editor. This setting defaults to unchecked.
`Font`	Allows the user to select the `Font`, `Font Style` and `Size` of the text editor.
`Color Theme`	Allows the user to select background and foreground colors of the text editor.

Table 14: Logic Editor Setting Parameters

Right Click Menu

When you right click on the logic editor window, the context menu with editor features such as Copy/Paste, Find/Replace, and Undo/Redo is displayed. Additionally, the settings in Advanced and Properties allow you to change the look of the editor.

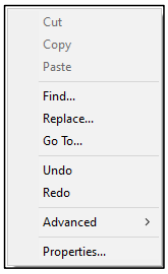


Figure 19: Right Click Menu

Menu Item	Description
Cut/Copy/Paste	`Cut` removes the selected text from the logic editor and copies it to the clipboard. `Copy` leaves the selected text in the logic editor and copies it to the clipboard. `Paste` inserts the text from the clipboard at the current cursor position.
Find/Replace/Go To…	`Find` displays the `Find` dialog, allowing you to use common search methods. `Replace` displays the `Find & Replace` dialog, allowing you to use common search and replace methods. `Go To…` displays the `Go To Line` dialog allowing you to go to the entered line number. <Ctrl>+<Alt>+<F3> will also display the dialog.
Undo/Redo	Undo the previous action. <Ctrl>+<Z> and <Alt>+<Backspace> are also supported. Redo the previous undo action. <Ctrl>+<Y> and <Ctrl>+<Shift>+<Z> are also supported.
Advanced	Displays the `Advanced` menu.
Properties…	Displays the `Window Properties` dialog.

Table 15: Right Click Menu

Advanced Menu

The Advanced menu contains less common features including allowing you to see Whitespace (spaces and tabs), convert selected text to Upper/Lower case, Comment/Uncomment selected logic, Sort lines Ascending/Descending, as well as display the Set Repeat Count and Choose Command dialogs.

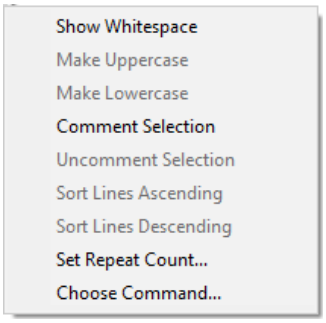


Figure 20: Advanced Menu

Menu Item	Description
Show Whitespace	Enables/disables whitespace marking. When enabled, spaces are shown as bullets (•) and tabs are shown as double arrows (»). When disabled, spaces and tabs are shown as empty space. <Ctrl>+<Alt>+<T> will toggle between enabled/disabled. The default setting is disabled.
Make Uppercase Make Lowercase	Makes the selected text all uppercase or lowercase. <Ctrl>+<Shift>+<U> will convert the text to all uppercase. <Ctrl>+<U> will convert the text to all lowercase.
Comment Selection Uncomment Selection	Either comments out the current line/selected lines or uncomments the current selection.
Sort Lines Ascending Sort Lines Descending	Sorts the selected lines in ascending or descending order.
Set Repeat Count	Displays the `Set Repeat Count` dialog. The repeat count dialog prompts the user for the number of times to repeat the next command.
Choose Command…	Displays the `Choose Command` dialog, which allows you to use all commands supported by the text editor.

Table 16: Right Click Menu

Window Properties Dialog

When you right click on the logic editor and select the `Properties…` menu option, the `Window Properties` dialog will be displayed. The `Color/Font` tab allows you to set the colors and fonts for various editor settings.

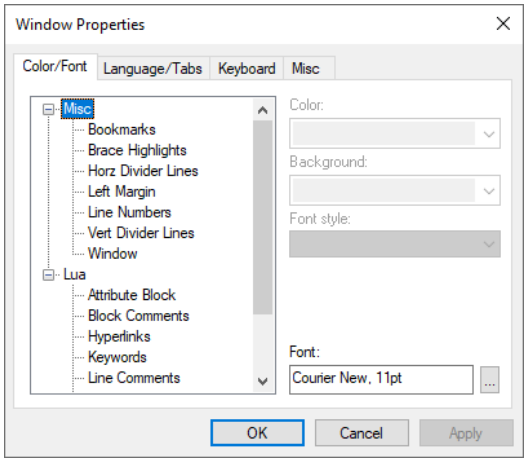


Figure 21: Properties Menu – Color/Font tab

The `Language/Tabs` tab allows you to configure the Lua language logic editor settings including adjusting case while typing, indentation, statement completion and tab parameters.

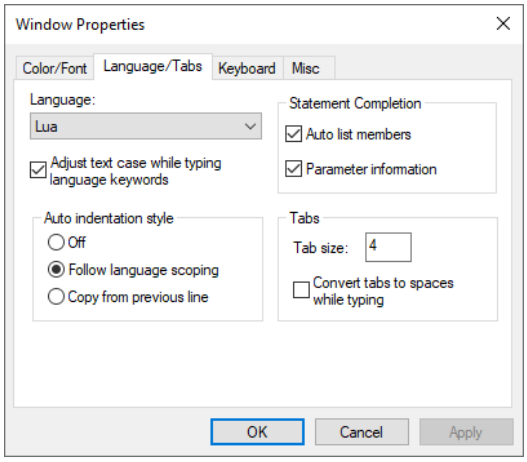


Figure 22: Properties Menu – Language/Tabs tab

The `Keyboard` tab allows you to view and modify the key assignments for editor commands.

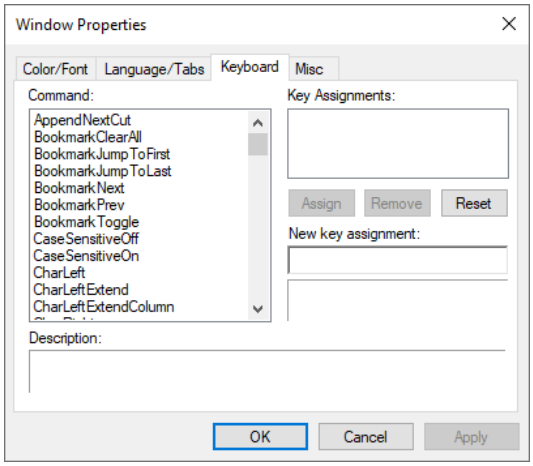


Figure 23: Properties Menu – Keyboard tab

The `Misc` tab allows you to set the following miscellaneous editor settings.

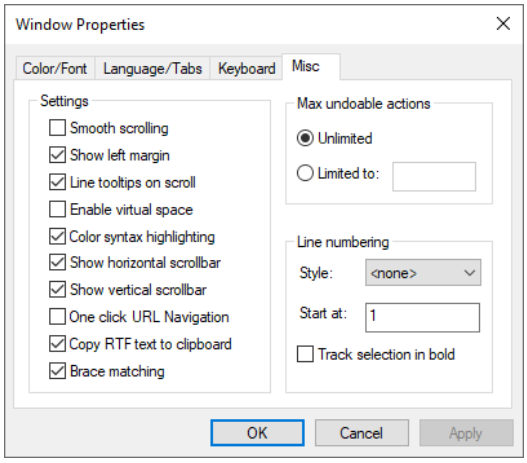


Figure 24: Properties Menu – Misc tab

V. Logic Simulator Tools

NCD includes a logic simulator to test code during development. The logic simulator allows you to start/pause code, view/force data values, set/remove breakpoints, step into/out of/over code and view call stack/variables. The simulator icons will appear when the `Simulate` button is clicked, which will start simulation, but break on the first executable line of code. Clicking the resume button will start running the code.

Figure 25: Logic Simulator Toolbar

The following table describes each of the toolbar features.

Feature	Tool Tip	Description
Simulate	Simulate	Starts logic simulation. The simulator will break at the first line of code in the `Load` (DefaultStartFunction) function to allow debug of the Load function. Clicking the `Resume` button will run the logic until it reaches a user breakpoint (if present).
End Simulate	End Simulate	Ends logic simulation. Simulator logic values will be reset to zero when the next simulation session is started.
	Toggle Spy List	Display or hide the Spy List. The Spy List allows you to drag in points from `Inputs`, `Outputs`, `Logic Inputs` and `Logic Outputs`. This can be usefully when working on logic that utilizes multiple points that cannot be displayed on the screen at the same time.
	Toggle Breakpoint	Toggles a breakpoint. After clicking on the desired line in the logic editor, click this button to add a breakpoint if no breakpoint is present or remove a breakpoint if a breakpoint is present.
	Remove all Breakpoints	Removes all breakpoints in logic.
	Resume	Starts or resumes logic solving. This button must be clicked when the Simulate button is pressed to start logic and after a breakpoint stops to resume logic solving.
	Break on next executable code	Breaks logic solving on the next line of executed code. Depending upon `Timer` and `DataChange` events, the break could cause logic solving to halt on a line of code immediately or as long as it takes for a `Timer` to run or a `DataChange` event to occur.
	Step Into Function	Steps into a function. When this button is clicked, the logic will advance to the next line of code. If the current line of code is calling a function, the next line will be in that called function. This button is only enabled when logic is stepping (at a breakpoint).
	Step Over Function	Steps over a function. When this button is clicked, the logic will advance to the next line of code in the current function, even if the current line of code is calling a function. This button is only enabled when logic is stepping (at a breakpoint).

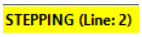
Feature	Tool Tip	Description
	Step Out of Function	Steps out of a function. When this button is clicked, the logic will continue execution of the current function until it returns to its caller. When using this button in a `Timer` or `DataChange` event, the code will continue to run until the next breakpoint (if present). This button is only enabled when logic is stepping (at a breakpoint).
	View Call Stack	Opens a dialog box that displays the call stack. This is an advanced debugging feature that displays the functions that are currently on the call stack by order of how they were called. The first function in the list is the currently executed function.
	View Variables	Opens a dialog box that displays global and local variables for the current function.
	Run	When `RUN` is displayed, the logic is currently running.
	Stepping Code	When `STEPPING` is displayed, the logic is at a breakpoint indicated by the line number.

Table 17: Logic Simulator Tools

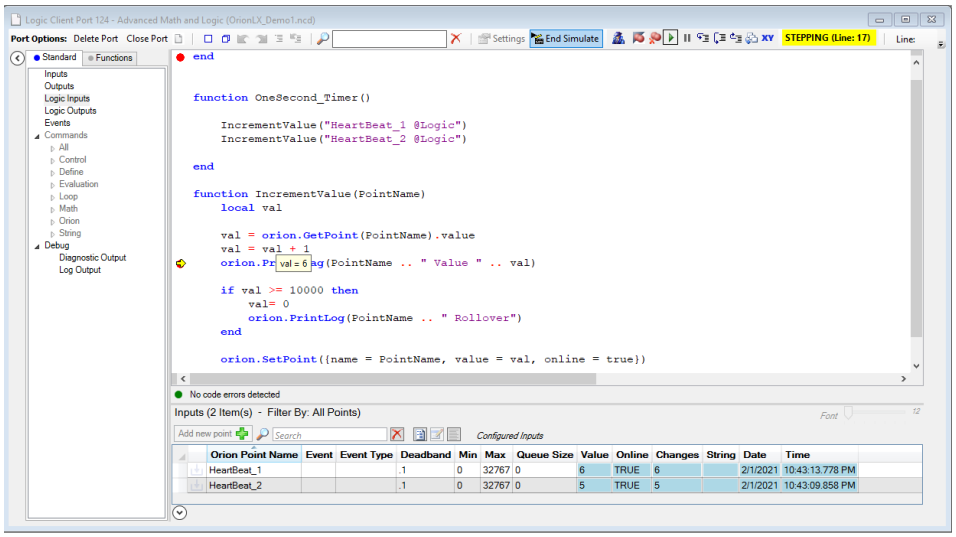


Figure 26: Logic Simulator

The figure above shows the `Advanced Math & Logic` window during simulation. In this screen capture, the logic is at a breakpoint on line 17 and the mouse is hovering over the variable `val`, which is displaying a value of 6. The value will only be displayed on mouse over when the logic is at a breakpoint. The `Logic Inputs` menu at the bottom is displaying the real-time values and attributes in blue background. The values and attributes can be forced by double clicking on any of the blue background attributes to bring up the `Force Point Values` dialog. When viewing multiple points that aren't in the same point type (`Input`, `Output`, `Logic Input`, `Logic Output`), the `Spy List` allows you to drag/drop points of interest.

`Debug` menus are available in the Standard tree on the left. The Debug menus allow you to view the `orion.PrintDiag` (`Diagnostic Output`) and `orion.PrintLog` (`Log Output`) output. The following figures provide a sample of each type of output. Buttons are provided to sort the output in ascending/descending order, copy contents to the clipboard, save contents to a file, delete contents, and to float the window to another location.

Diagnostic Output

The simulator provides the following diagnostic output.

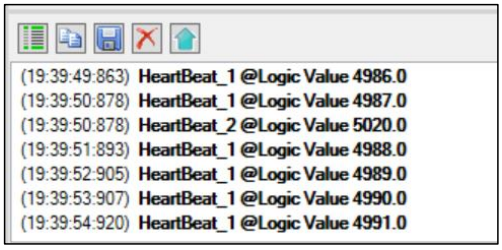


Figure 27: Logic Simulator Diagnostic Output

Log Output

The simulator provides the following log output.

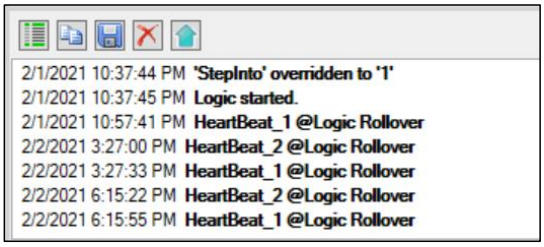


Figure 28: Logic Simulator Log Output

Button	Tool Tip	Description
	Toggle showing newest items at top or bottom of list	Sorts the output in ascending or descending order.
	Copy contents of log to clipboard	Copies the entire contents of the output to the clipboard.
	Save contents of log to a file	Saves the entire contents of the output to a file.

Button	Tool Tip	Description
	Delete contents of log window	Deletes the entire contents of the output.
	Float window	Floats the window so it can be moved to another location. The window will no longer be constrained to the NCD container. Both the `Diagnostic Output` and `Log Output` can be floated. To return the output to the NCD container, click the down arrow ().

Table 18: Logic Simulator Diagnostic and Log Tools

Spy List

The `Spy List` allows you to view specific data values for the logic you are testing. This is useful when the data points are on different menus (`Input`, `Output`, `Logic Input`, `Logic Output`) or if the points cannot be sorted to display without scrolling. To add points to the `Spy List`, drag and drop from the appropriate menu. Real-time attributes are displayed with a blue background. Attributes can be forced by double clicking on any of the blue background attributes to bring up the `Force Point Values` dialog. To remove a point from the `Spy List`, click the red X next to the `Orion Point Name`.

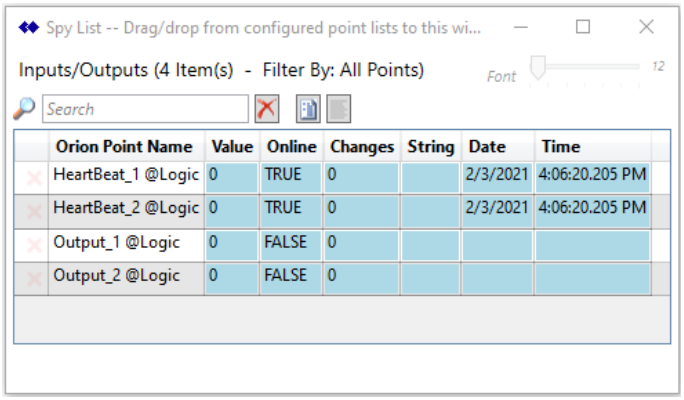


Figure 29: Logic Simulator Spy List

Forcing Values and Attributes

Values and attributes can be forced to test logic. To force, double click on any value or attribute with a blue background. The following dialog will display, allowing you to force multiple values or attributes at one time. Check the box next to the desired values and attribute(s) to force. Depending on the `Timed` checkbox, values and attributes will either be forced for a duration or a single shot. Logic cannot write to values or attributes when they are time forced. Outputs and Logic Outputs cannot be time forced since they are always single shot.

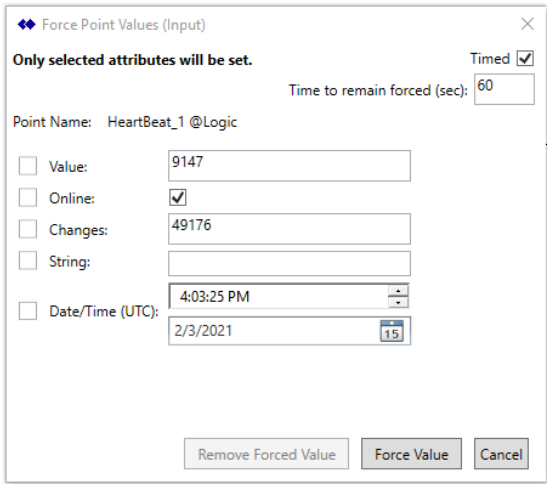


Figure 30: Logic Simulator Force Point Values Dialog

Feature	Description
Timed	Enables a timed forced attribute(s). After the `Time to remain forced` time, the attribute(s) will return to the contents before the force. Attributes that have been forced will display with a light green background. Logic cannot write to attributes while a point is time forced. Only present for Inputs and Logic Inputs, since outputs are always forced untimed.
Time to remain forced (sec)	When the `Timed` checkbox is checked, this sets the amount of time the attribute(s) will remained forced before returning to the contents before the force. Logic cannot write to attributes while a point is time forced. Only present for Inputs and Logic Inputs, since outputs are always forced untimed.
Value	When the checkbox to the left of `Value` is checked, the logic simulator will force the value attribute with the contents of the textbox to the right when the **Force Value** button is pressed.
Online	When the checkbox to the left of `Online` is checked, the logic simulator will force the online status attribute with the contents of the checkbox to the right when the **Force Value** button is pressed.
Changes	When the checkbox to the left of `Changes` is checked, the logic simulator will force the changes attribute with the contents of the textbox to the right when the **Force Value** button is pressed.
String	When the checkbox to the left of `String` is checked, the logic simulator will force the string attribute with the contents of the textbox to the right when the **Force Value** button is pressed.
Date/Time (UTC)	When the checkbox to the left of `Date/Time (UTC)` is checked, the logic simulator will force the Date/Time attribute with the contents of the boxes to the right when the **Force Value** button is pressed.

Table 19: Logic Simulator – Force Point Value

VI. Logic Examples

In this chapter, examples of several Advanced Math & Logic applications, including their setup, are provided. The main chapter topics are as follows.

Single Comm Fail Point for Multiple Field Devices

When the Orion polls multiple field devices and the polled data is used by the Orion or a client station in a specific calculation, a communication error to any one of the devices will result in an invalid result for the overall calculation.

This logic monitors the Comm Fail points from two devices. As long as both devices are communicating, indicated by a 0 value, the combined Comm Fail flag will indicate good communications with a 0 value. If either of the devices (or both) are not communicating, indicated by a 1 value, the combined Comm Fail flag is set to a 1 value, indicating an error.

This example will explain the steps necessary to `OR` multiple Comm Fail points into a logic input using a timer. The logic will be executed every 1000ms and set the combined Comm Fail point to either 0 or 1.

Step 1: Create the Logic Timer Event (Figure 31)

- Under the `Events` menu, type CommFailEvent in the `Create New Event` textbox and click the **`Create`** button to create a new event.
- From the `Event Type` dropdown list, select `Timer`.
- Set the `Timer Interval` to 1000 milliseconds. This value determines how often the function (i.e. `CommFailEvent_Timer`) will be executed.
- Set the `Timer Enabled` to `True` to enable the timer event.

The following window will be displayed (Figure 31). Notice the new `CommFailEvent_Timer` () function has been added to the logic editor. The code for `OR`ing the Comm Fail points will be placed into this function.

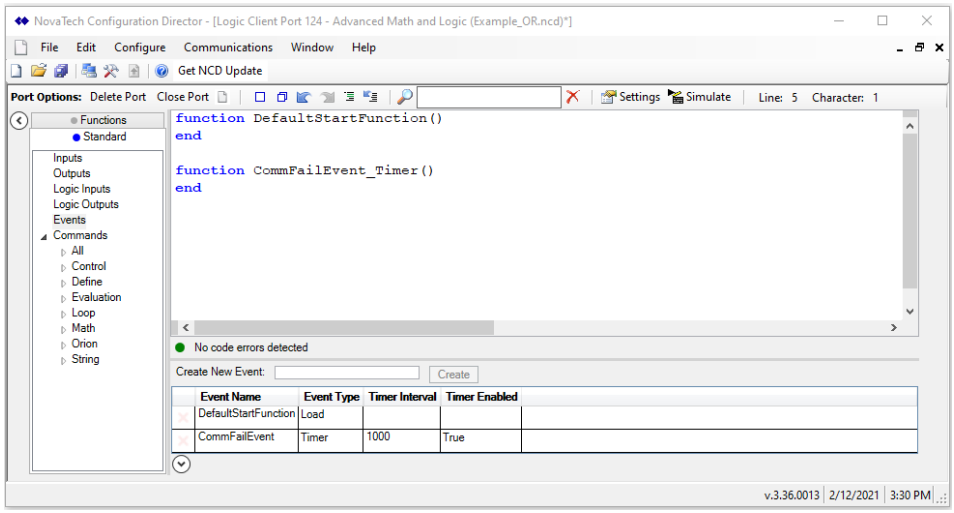


Figure 31: Create the CommFailEvent

Step 2: Select Comm Fail Points of Field Devices (Figure 32)

The next step is to include both Comm Fail points of the respective field devices into the logic. Under the `Inputs` menu, (Figure 32), select the Comm Fail points for the respective field devices from the `Point Name List` and copy them to the `Configured Inputs` list.

Note that the `Event` field is left blank. Since a timer-based event is used to monitor the Comm Fail points, no data change event needs to be associated with the Comm Fail points.

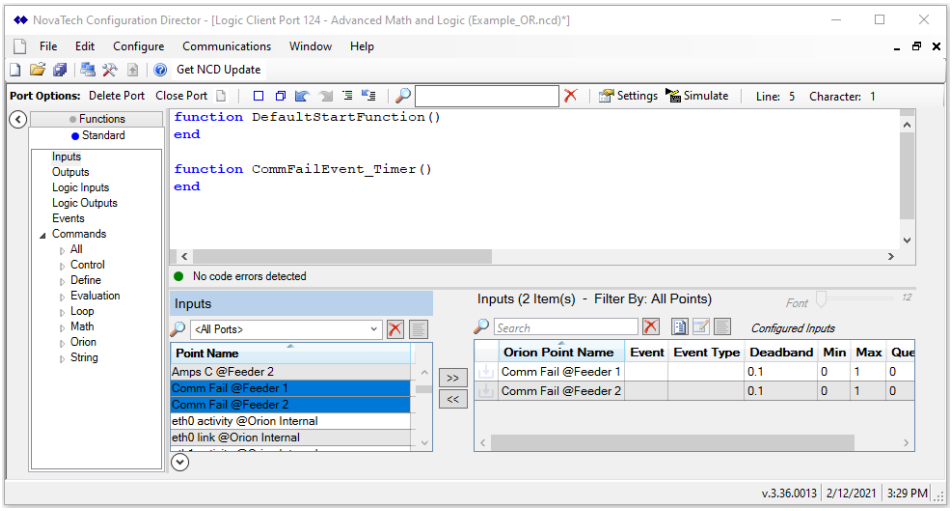
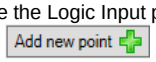


Figure 32: Select the Comm Fail Points

Step 3: Create the logic input point that will store the result.

Under the `Logic Inputs` menu, create the Logic Input point for storing the result of the logic (Figure 33).

Under the `Logic Inputs` menu, click 

to create a new logic input point in the list. Type `RelayCommFail` for the Orion `Point Name`.

Note that the `Event` field is left blank for this point as well.

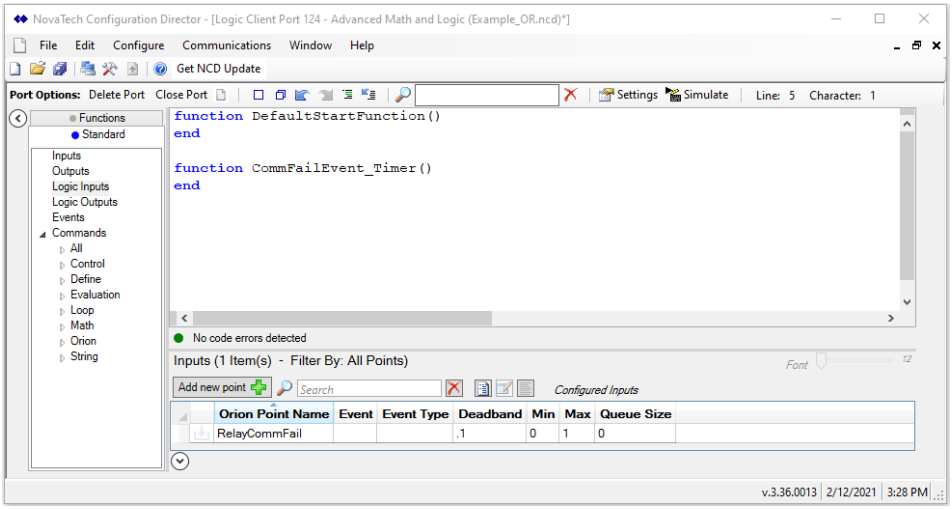


Figure 33: Create Logic Input Point for Result

All input points required for the Comm Fail logic are now configured.

Step 4: Writing the code to OR the Comm Fail points (Figure 34)

The logic obtains the value of the Comm Fail point for both devices (`Comm Fail @Feeder 1` and `Comm Fail @Feeder 2`) using the **`orion.GetPoint`** function. If either point (or both) indicates a communications failure, the overall Comm Fail point (`RelayCommFail @Logic`) will be set to 1 using the **`orion.SetPoint`** to indicate a failure. If both points indicate good communications, `RelayCommFail @Logic` will be set to 0.

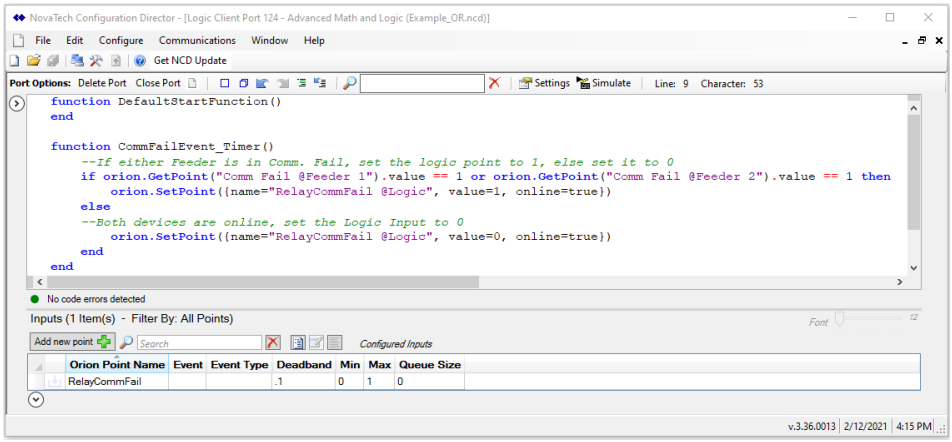


Figure 34: Setting up Comm Fail Logic

Test and debug using the Logic Simulator integrated into the logic editor.

Note that in order to save logic to the .lua file, `Save` or `Save as` from the NCD main menu must be used.

Step 5: Map `RelayCommFail @Logic` to a server port

`RelayCommFail @Logic` (which now indicates the overall communications status) can be mapped to any server port on the Orion as needed. Once mapped, any client station connected to any of those server ports can obtain `RelayCommFail @Logic`, and thus a summary of the communication status of the field devices.

Automatic Switchover Between Values from Primary and Secondary Devices

When redundant devices are used for critical feeders, logic can be used to select which device's data is provided to SCADA. Under normal circumstances, the Orion uses the primary device's data. If the primary device goes offline, the Orion needs to use the secondary device's data. If both devices are offline, the data is set to zero and the points are marked offline. The logic running in the Orion will make these determinations as follows.

1) The logic checks the Comm Fail status for both the primary device and the secondary device.
2) If the primary device's Comm Fail point indicates the device is online, the logic copies the primary device's data to logical inputs. It also sets these logic inputs as being online so they are indicated as valid to the client station.
3) If the primary device's Comm Fail point indicates offline, and the secondary device's Comm Fail indicates online, then logic copies the secondary device's data to the same logical inputs. It also sets these logic inputs as being online so they are indicated as valid to the client station.
4) If both device's Comm Fail points indicate offline, then the logic sets all logical inputs to zero, and sets their communication status to offline. From the Comm Fail points, the client station can also see that both the primary device and the secondary device are offline.

Step 1: Create the Logic Timer Event (Figure 35)

- Under the `Events` menu, type an event name in the `Create New Event` textbox and click the **`Create`** button to create a new event.
- From the `Event Type` dropdown list, select `Timer`.
- Set the `Timer Interval` to 1000 milliseconds. This value determines how often the function (i.e. `CheckFeeder1Switchover_Timer`) will be executed.
- Set the `Timer Enabled` to `True` to enable the timer event.

The following window will now be displayed (Figure 35). Notice the new `CheckFeeder1Switchover_Timer` function has been added to the logic editor. The logic code needed for the automatic switchover will be placed into this function.

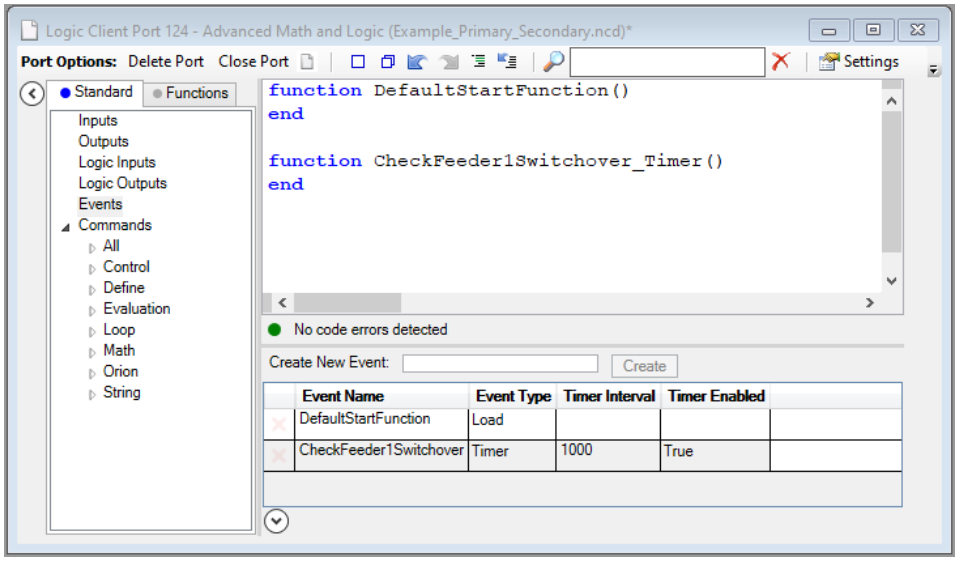


Figure 35: Create Event for Automatic Switchover

Step 2: Select the redundant input points from both the primary and secondary devices as logic inputs.

Under the `Inputs` menu, (Figure 36), select the redundant input points as well as the Comm Fail points from both the primary and secondary devices in the `Point Name` list, and copy them to the `Configured Inputs` list by clicking on **>>**. The `Min` and `Max` will default to the range defined at the client port which polls the devices.

Note that for all points, the `Event` field is left blank. Since a timer-based event is being used to monitor the Comm Fail points, association to a data change event to the Comm Fail points is not required.

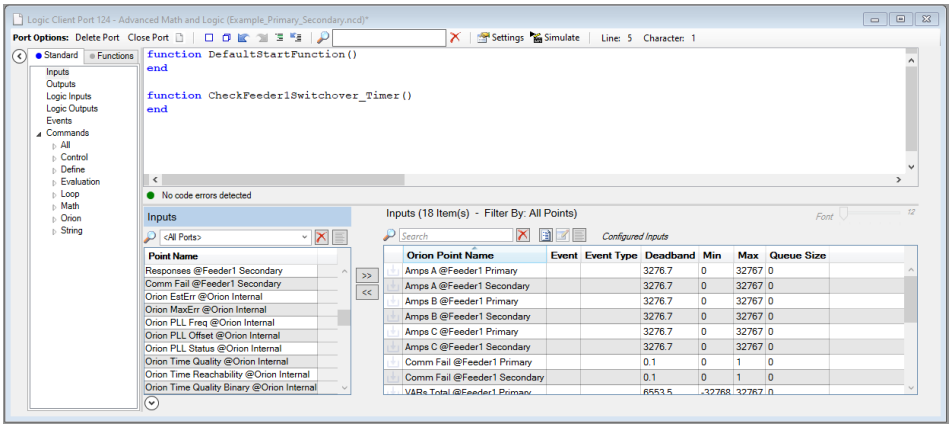


Figure 36: Select Primary/Secondary Input Points

Step 3: Create the logic input points to which the redundant input points will be copied.

Under the `Logic Inputs` menu, (Figure 37), create an appropriately named point for every pair of redundant input points. Depending on which one of the two redundant input points is online, the logic will copy one of the input points to the respective logic input point.

Under the `Logic Inputs` menu, click 

to create a new logic input point in the list. Fill in the `Orion Point Name`. The `Event` field must be left blank (none of these points will trigger an event).

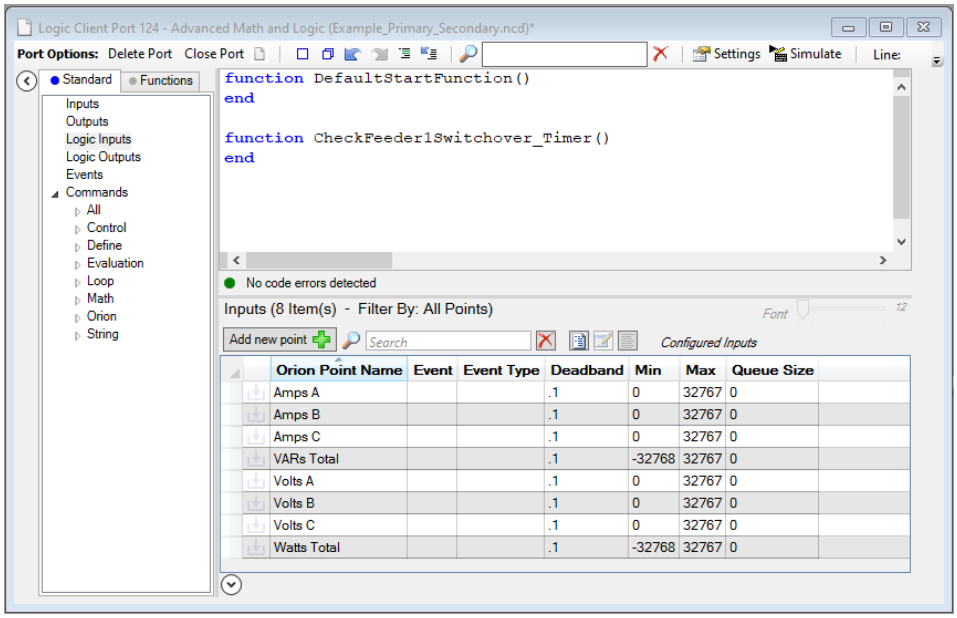


Figure 37: Create Logic Input Point List

Now that all points required for the logic have been created, the next step is to write the actual logic.

Step 4: Write logic to determine the live device and set the logic input points.

The logic consists of the following steps (Figure 38).

- The logic must first obtain the Comm Fail points for both devices.
- If the primary device is online, the logic copies the device's input points to the logic input points, and sets the logic input points to online.
- If the primary device is offline and the secondary device is online, the logic copies the input points from the secondary device to the logic input points, and sets the logic input points to online.
- If both devices are offline, the logic copies 0 to the logic input points and sets the logic input points to offline.

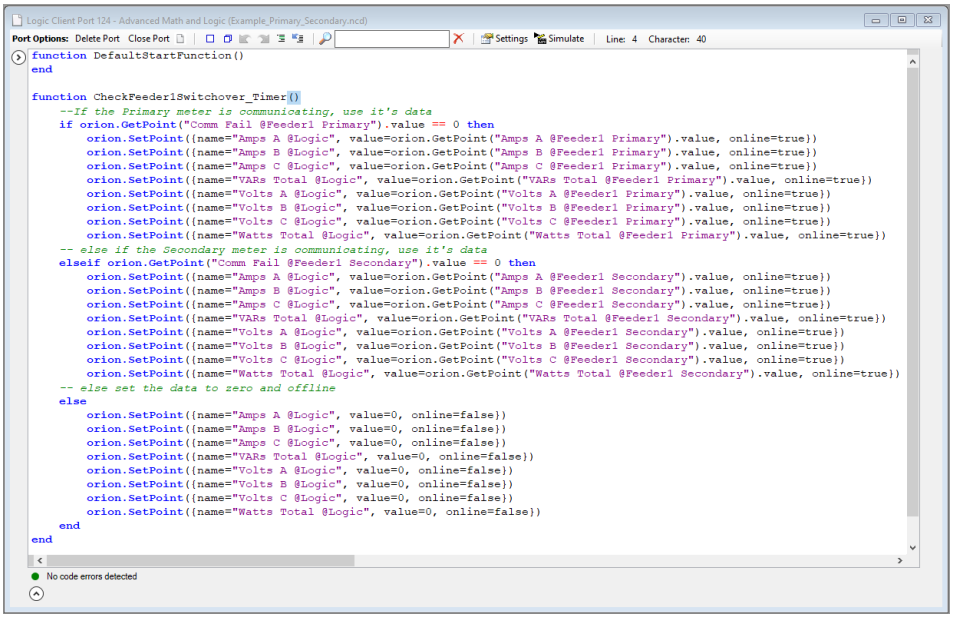


Figure 38: Automatic Switchover Logic

Test and debug using the Logic Simulator integrated into the logic editor.

Step 5: Map the points to server ports.

The `Logic Inputs` can be mapped to any server port on the Orion as needed. Once mapped, any client station connected to any of those server ports can obtain the data coming from the primary or secondary devices based on communication status.

Note that in order to save logic to the .lua file, `Save` or `Save as` from the NCD main menu must be used.

Simple Calculation

When using older IEDs, it may be necessary to compute the 3 Phase Current from the A phase, B phase, and C phase values read from a device. The following example computes 3 Phase Current every time one or more of the phase values change.

Step 1: Create the DataChange event (Figure 39)

- Under the `Events` menu, type Feeder1_Amps3P in the `Create New Event` textbox and click the **`Create`** button to create a new event.
- From the `Event Type` dropdown list, select `DataChange`.

The following window will be displayed (Figure 39). Notice the new `Feeder1_Amps3P()` function has been added to the logic editor. The logic code for the phase current calculation will be placed into this function.

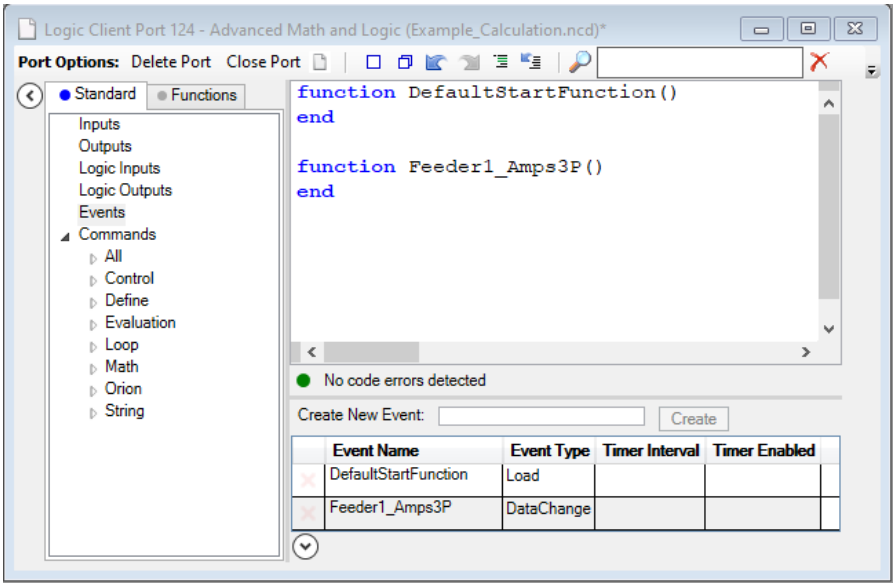


Figure 39: Create Event for Calculation

Step 2: Select the A phase, B phase, and C phase points as inputs.

Under the Inputs menu, (Figure 40), select the phase current input points in the `Point Name` list, and copy them to the `Configured Inputs` list by clicking **`>>`**. Note that since the function `Feeder1_Amps3P()` should be executed every time one or more of the three values change, each input point must be set up with `Feeder1_Amps3P` as `Event`. Select the `Deadband` event type and change the deadband to 1 for all three points so the event will trigger when any of the three points change more than a value of 1.

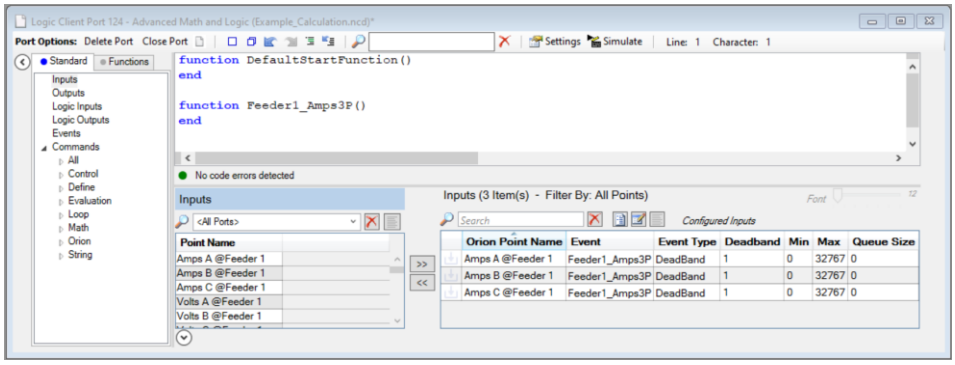


Figure 40: Select Phase Current Input Points

Step 3: Create the point to which the result of the phase current calculation will be copied.

Under the `Logic Inputs` menu (Figure 41), create a point for storing the result of the phase calculation by clicking on Add new point to create a new row in the list. Fill in the `Orion Point Name`. Then configure the `Min` and `Max`. Since this point is an analog point derived from analog inputs from the field, it should have the same `Min` and `Max` as the field inputs. The `Event` field must be left blank (computing the phase current does not trigger another event).

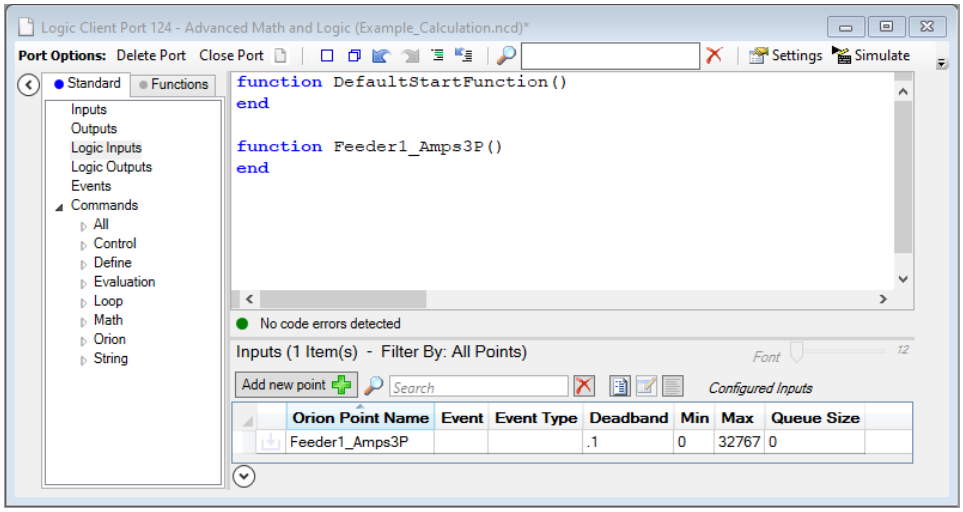


Figure 41: Create Logic Point List

Now that all points required for the logic have been set up or created, the next step is to write the actual logic.

Step 4: Write logic for calculating the phase current (Figure 42)

This logic will be executed every time one of the three phase inputs from the field changes.

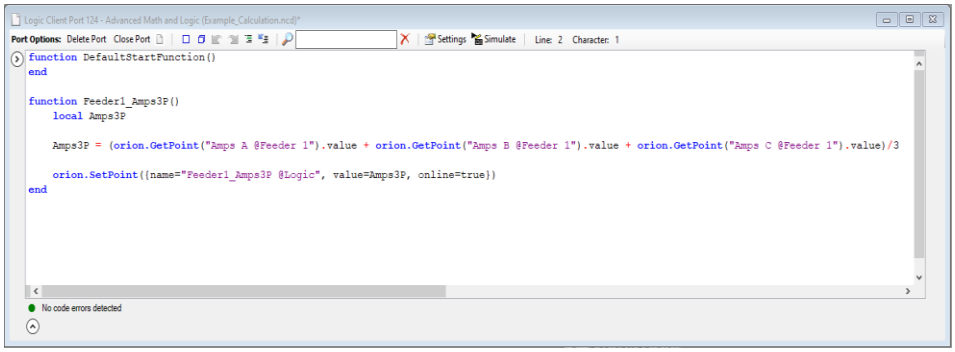


Figure 42: Phase Calculation Logic

Step 5: Map `Feeder1_Amps3P @Logic` to any server port on the Orion as needed.

Once server ports are mapped, any client station connected to any of those server ports can obtain the three phase values from the device, as well as the result of the calculation.

Time-Delayed Output Operation

This logic monitors a discrete input status to determine whether a level has been reached. When the field input status changes to 1, the level has been reached and the logic starts a timer. After 5 seconds, the logic sends the output command 1 to the device. When the input status changes to 0, the logic immediately sends the output command 0 to the field device without delaying.

Step 1: Create the logic events (Figure 43)

IsReached DataChange event:

- Under the `Events` menu, type `IsReached` in the `Create New Event` textbox and click the **`Create`** button to create a new event.
- From the `Event Type` dropdown list, select `DataChange`. The event type is is the only parameter that needs to be set up.

Shutoff Timer event:

- Under the `Events` menu, type `Shutoff` in the `Create New Event` textbox and click the **`Create`** button to create a new event.
- From the `Event Type` dropdown list, select `Timer`.
- Set the `Timer Interval` to 5000 milliseconds. This value determines how often the function (i.e. `Shutoff_Timer`) will be executed.
- Set the `Timer Enabled` to `False` to disable the timer event as shown below (Figure 43). This timer will only run when enabled in logic.

The following window will be displayed.

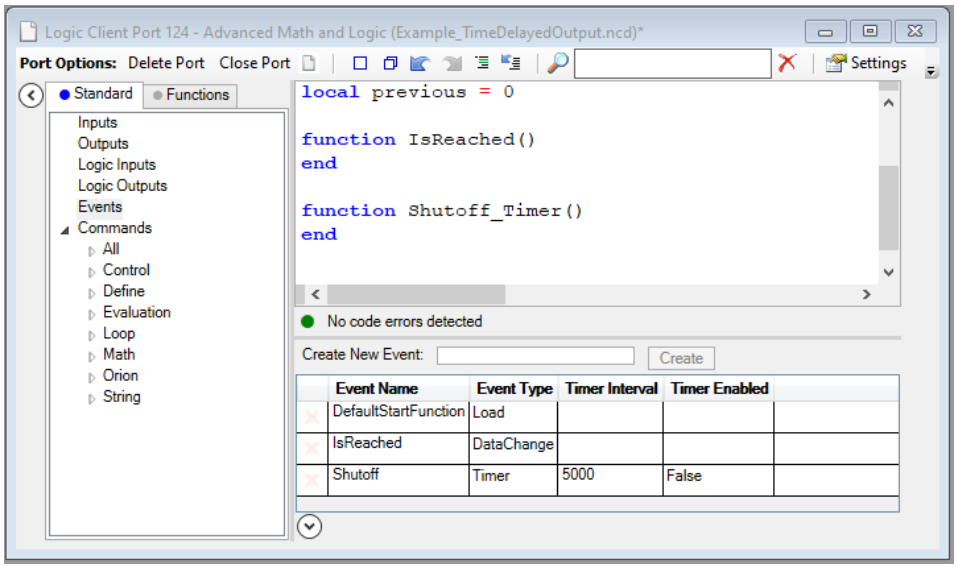


Figure 43: Create Events for Shutoff Logic

Step 2: Select `LevelReached @I/O Device` from Inputs.

Under the `Inputs` menu, (Figure 44), select `LevelReached @I/O Device` in the `Point Name` list, and copy it to the `Configured Outputs` list by clicking **>>**. Note that since the function `IsReached` should be executed every time `LevelReached @I/O Device` changes, `LevelReached @I/O Device` must be set up as the event for `IsReached`. Select the `Deadband` event type and change the deadband to 0.1 so the event will trigger when the point changes more than a value of 0.1.

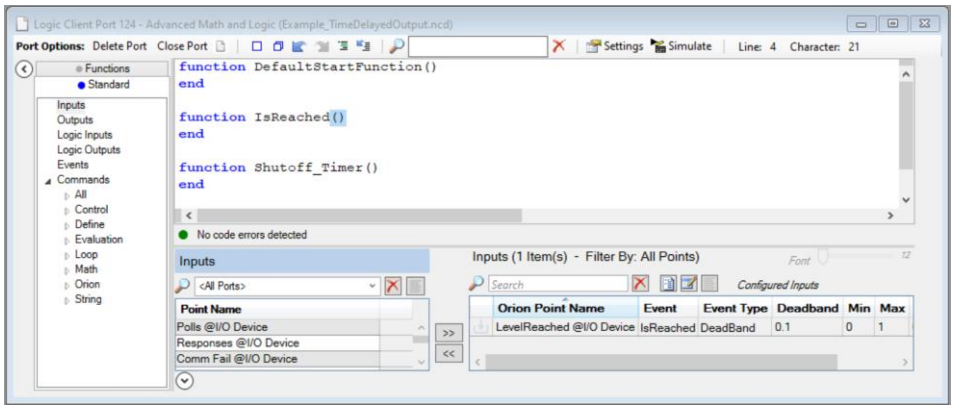


Figure 44: Select Device Input Point

Step 3: Select `Shutoff @I/O Device` from Outputs.

Under the Outputs menu, (Figure 45), select `Shutoff @I/O Device` in the `Point Name` list, and copy it to the `Configured Outputs` list by clicking **>>**. The `Event` field must be left blank.

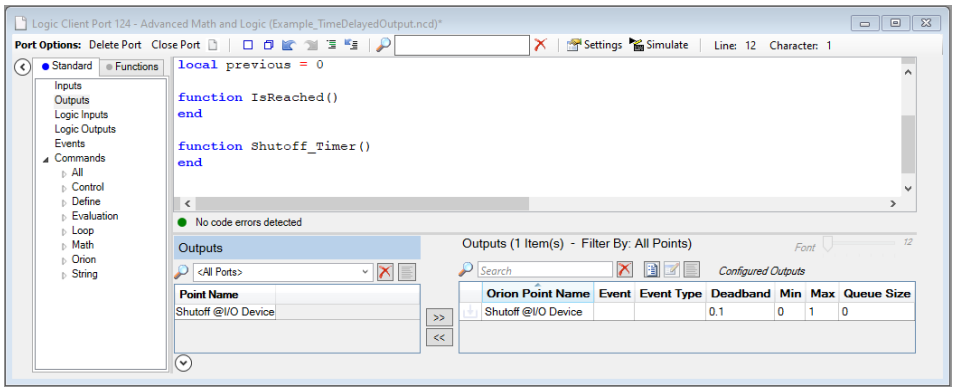


Figure 45: Select Device Output Point

Now that all points required for the logic have been set up or created, the next step is to write the actual logic.

Step 4: Write logic for triggering the shutoff commands (Figure 46)

The routine `IsReached` is to be executed every time `LevelReached @I/O Device` is updated. Typically, the value is updated when the client port polls the point `LevelReached @I/O Device` from the device. If `LevelReached @I/O Device` changes to 1, `Shutoff @Device` will be set to 1 after a delay of 5 seconds. If `LevelReached @I/O Device` changes to 0, `Shutoff @I/O Device` will be set to 0 immediately.

The timer `Shutoff` is enabled only if the previous value of `LevelReached @I/O Device` is 0, and the current value of `LevelReached @I/O Device` is 1. If the previous value is 1, and the current value is 0, the shutoff command is cleared immediately.

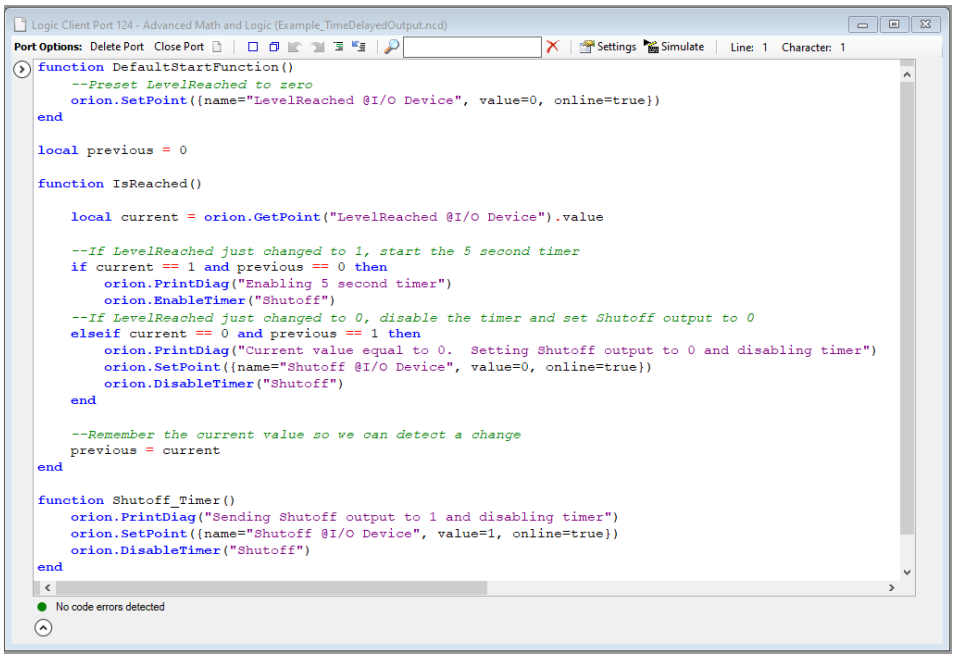


Figure 46: Shutoff Logic

Appendix A – List of Advanced Math & Logic Commands

Programming Elements

The Orion Advanced Math & Logic supports the following programming elements.

- Variable declarations
- Conditional statements
- Repetitive statements (loops)
- Functions

Orion Advanced Math & Logic utilizes the Lua programming language (Lua 5.3.5 – Copyright © 2015–2020 Lua.org, PUC-Rio). For a complete reference of the Lua programming language, refer to the complete online reference manual via the following link: http://www.lua.org/manual/5.3/manual.html.

`local` Variable Declaration

Local variables are defined with `local` inside of a function. Any variable is assumed to be global unless explicitly declared inside a function as a local. Global variables do not need to be declared, but can be declared using `local` outside of a function. Before the variable is used on the right side of an = assignment, it must be assigned a value. At the end of a function, local variables are not maintained to the next execution of the same function. Global variables store the value between two executions of the function and can be used by any function

Syntax:

```
local variable_name
```

Example:

```
local i

--now i can be used
i = orion.GetPoint("Breaker Status @Feeder1")
```

`if` Conditional Statement

The `if` statement allows specific code to be executed only if a certain condition or sets of conditions are met. The conditions are specified with expressions (e.g. `i > 5` or `a == 1 and b ~= 1`). Expressions are specified using Logic Operations to compare variables and values.

Syntax:

```
if expression then
  <list of statements>
[elseif expression then
  <list of statements>]
[else
  <list of statements>]
end
```

Example:

```
local i

--Returns the Value of "Point1 @Device1"
i = orion.GetPoint("Point1 @Device1")
if i.value > 50 then
 --Do something here
elseif i.value > 25 then
 --Do something here
else
 --Do something here
end
```

`for` Loop

The `for` statement allows specific code to be repeatedly executed. The number of repetitions is specified ahead of time. A running counter is incremented for every repetition of the loop. The running counter can be incremented at every repetition by a specific increment specified in `StepValue`. If no increment value is specified, the running counter will be incremented by 1.

Syntax:

```
for <CounterVariable> = <InitialValue>, <EndValue> [, <StepValue>] do
  <list of statements>
end
```

Example:

```
local i
--This will print 2 4 6 8 10 12 14 16 18 20 to the output window
for i = 1, 10 do
  orion.PrintDiag(i * 2)
end
```

`while` Loop

The `while` statement allows specific code to be repeatedly executed when the specified condition is met. The condition is checked before the first iteration of the loop and is repeated with every iteration until a specified condition no longer met. A while loop will not execute, if the condition is not met. The conditions are specified with expressions (e.g. `i > 5` or `a == 1 and b ~= 1`). Expressions use Logic Operations to compare variables and values.

Syntax:

```
while expression do
  <list of statements>
end
```

Example:

```
local i

--This will print 20  30  40  50
i = 10
while i < 50 do
  i = i + 10
  orion.PrintDiag(i)
end
```

`repeat` Loop

The `repeat` statement allows specific code to be repeatedly executed. The condition is checked after the first iteration of the loop and is repeated with every iteration until a specified condition is no longer met. A `repeat` loop will always execute at least once. The conditions are specified with expressions (e.g. `i > 5` or `a == 1 and b ~= 1`). Expressions use Logic Operations to compare variables and values.

Syntax:

```
repeat
  <list of statements>
until expression
```

Example:

```
local i

--This will print 20  30  40  50
i = 10
repeat
  i = i + 10
  orion.PrintDiag(i)
until i >= 50
```

`function` Function

If the same code, such as a math formula, is required in multiple locations in the logic, it is easier to encapsulate that code once as a function, and then call the function every time that formula needs to be computed. Thus, the `function` statement reduces overall code length and complexity. A `function` can be tied to an event and executed when that event occurs (see Events Menu section for setup details). In this case the `function` does not return results to the calling location.

Syntax:

```
function FunctionName (parameter {, parameter})
  <list of statements>
  return result
end
```

Parameters:

Zero or more parameters can be specified.

Return Value:

The return statement is used as an option to assign the result(s) of the function. The resulting value(s) are returned to the calling location.

Examples:

```
function GetStatusName(x)
  local s
  if ( x == 1 ) then
    s = "Meter1Status @Logic"
  elseif ( x == 2 ) then
    s = "Meter2Status @Logic"
  elseif ( x == 3 ) then
    s = "Meter3Status @Logic"
  elseif ( x == 4 ) then
    s = "Meter4Status @Logic"
  end
  return s
end


function Circumference(radius)
  return 2 * radius * 3.14159265
end
```

```
function MySub()
  local sVar
  local i

  --This will print the values for database points
  --"Meter1Status @Logic", "Meter2Status @Logic",
  --"Meter4Status @Logic", and '"Meter4Status @Logic"

  for i = 1, 4 do
    sVar = GetStatusName(i)
    i = orion.GetPoint(sVar)
    orion.PrintDiag(i)
  end

  --This will print "12.5663706"
  orion.PrintDiag( Circumference (2) )

end
```

Database Operations

orion.GetPoint

Returns a specific point attribute or table of point attributes of the specified real-time database point. These attributes include:

`name`	String with name of the point. Example: "`Breaker Status @Feeder1`"
`index`	Number of index in the real-time database.
`value`	Value of the point.
`changes`	The number of value changes of the point.
`online`	The online status of the point. false indicates the point is Offline true indicates the point is Online
`string`	The optional string attribute of the point. nil if the string attribute hasn't been written to.
`time`	The timestamp of the last write of the point to the real-time database in ISO format in UTC time. The time is updated with every write, regardless of value changes. Example time: "2019-09-17 22:06:30.807Z"

Syntax: `orion.GetPoint(name)`

Parameters:

`name`	Name of the point whose table of point attributes is to be obtained. Example: `Breaker Status @Feeder1`.

Return Value: The specific attribute or a table with current numeric value, number of changes, online/offline status, time and associated string (if applicable), of the specified point.

Example:

```
local i = {}

--Returns a table of current attributes of
--"Breaker Status @Feeder1"
--i.value or i["value"] contains the point's value
i = orion.GetPoint("Breaker Status @Feeder1")
orion.PrintDiag("Breaker Status is " .. i.value)

local i

--Returns only "Breaker Status @Feeder1" value
i = orion.GetPoint("Breaker Status @Feeder1").value
orion.PrintDiag("Breaker Status is " .. i)
```

orion.SetPoint

Sets the `value`, `changes`, `online` status, associated `string`, `duration`, `offtime`, and/or `pulses` of the specified real-time database point. The point can be specified either by `name` or by `index`. `orion.SetPoint` will also automatically generate a change notification to inform other interested modules of the point's change.

Syntax: `orion.SetPoint(PointAttributes)`

Parameters: `PointAttributes` table entries:

`name`	String with name of the point whose attributes are to be set. Example: "`Breaker Status @Feeder1`"
`index`	Number with index of the point whose attributes are to be set. This index can be retrieved from a prior call to `orion.GetPoint`.
`value`	New value that the point will be set to. This value should be within the set scaling range for the point. However, setting the value below or beyond the set scaling is not restricted.
`changes`	This parameter is optional and sets the number of changes for the point.
`online`	This parameter is optional and sets the point's online status if desired. false sets the point to Offline true sets the point to Online
`time`	This parameter is optional and sets the timestamp of the point in ISO format in UTC time. Example time: "2019-09-17 22:06:30.807Z"
`duration`	This parameter is optional and sets the point's control duration in milliseconds. This parameter will be used in conjunction with a Trip/Close point operation.
`offtime`	This parameter is optional and sets the point's time between pulses in milliseconds. This parameter will be used in conjunction with a Trip/Close point operation.
`pulses`	This parameter is optional and sets the point's number of pulses. This parameter will be used in conjunction with a pulse train point operation.
`string`	This parameter is optional and sets the point's string attribute.

Return Value: None.

Example:

```
--Set the value of the point to 500 and its status
--to Online
orion.SetPoint({name="kWh    Phase    A    @Feeder1",
value=500, online=true})

--Set the value of the point to 500 without changing
--its online status
orion.SetPoint({name="kWh    Phase    A    @Feeder1",
value=500})
```

---

orion.AssocPoint

Associates the specified point with the specified data change function. The point must be specified by name, and the function argument specifies the function to be called when a data change event occurs. An optional object argument specifies the table which contains the function. This function can only be called in the Load function.

Syntax:

```
orion.AssocPoint(PointName, Function [, Object,
DataChangeType, Deadband])
```

Parameters:

`PointName`	String with name of the point whose associated data change function is to be set. Example: "`Breaker Status @Feeder1`"
`Function`	Function to be called when a data change event occurs on the associated point.
`Object`	Table containing the function. The default is a normal, global function if this optional argument is missing or `nil`.
`DataChangeType`	String indicating the data change type of either `refresh` or `deadband`. The default is `refresh` if this optional argument is missing or `nil`.
`Deadband`	Deadband value associated with `Deadband` data change type. This parameter is only meaningful when `DataChangeType` is specified as `Deadband`. The default deadband value is 0.0000000001 if this optional argument is missing.

Return Value: None.

Example:

```
--Associate the point with the kwh_a method in object
feeder1
orion.AssocPoint("kWh Phase A @Feeder1",
feeder1.kwh_a, feeder1, "deadband", 1)
```

orion.GetQueuedPoint

Returns a table of point attributes of the specified point's change event. These attributes include:

`name`	String with name of the point. Example: "`Breaker Status @Feeder1`"
`index`	Number of index in the real-time database.
`value`	Event value of the point.
`changes`	Event number of value changes of the point.
`online`	Event online status of the point. false indicates the point is Offline true indicates the point is Online
`string`	Event string attribute of the point. nil if string attribute was not written to.
`time`	Event timestamp in ISO format in UTC time. Example time: "2019-09-17 22:06:30.807Z"

Syntax: `orion.GetQueuedPoint(name)`

Parameters:

`name`	Name of the point whose table of event attributes is to be obtained. Example: "`Breaker Status @Feeder1`".

Return Value: A table of queued event attributes including numeric value, number of changes, online/offline status, time and string. If no events are queued for this point, nil will be returned.

Example:

```
local pt1 = {}

-- Get the first queued point change from the queue
pt1 = orion.GetQueuedPoint("Point1 @Device1")

-- If nil, no events
while pt1 ~= nil do

   orion.PrintDiag("Value: " .. pt1.value)
   -- Get the next queued event
   orion.GetQueuedPoint("Point1 @Device1")

end
```

Logging Operations

---

orion.PrintDiag

Prints the specified string to the `View Communications – Logic` Window of the Orion MMI, or the Diagnostic Output window of the Logic Simulator. Carriage return/line feed is automatically appended.

Syntax: `orion.PrintDiag(String)`

Parameters:

`String`	String to be printed to communication window of the Orion, or the Diagnostic Output window of the Logic Simulator. It is always printed on a new line.

Return Value: None.

Example:

```
--Prints "The value is 32"
orion.PrintDiag("The value is 32")
```

---

orion.PrintLog

Prints the specified string to the `System Menu – Event Log` window of the Orion MMI, and the Log Output window of the Logic Simulator. Carriage return/line feed is automatically appended.

Note: Excessive use can fill the Orion Event Log with entries to the point that the log becomes too large to extract meaningful information.

Syntax: `orion.PrintLog(String)`

Parameters:

`String`	String to be printed to the event log of the Orion, or the Log Output window the Logic Simulator. It is always printed on a new line.

Return Value: None.

Example:

```
--Prints "The value is 32" to the event log
orion.PrintLog("The value is 32")
```

Logic Operations

==
>
>=
<
<=
~=

Compare the values of two numbers, variables, or expressions.

Syntax: `operand operator operand`

where
- the operand can be a number, a variable, or an expression
- the operator is one of the following: ==, >, >=, <, <=, ~=

Parameters: All comparison operators are binary operators, i.e. they require a constant, variable, or expression on the left side and on the right side.

Return Value: Result of the comparison operation.

Example:

```
local i = {}

i = orion.GetPoint("Tap Position @Transformer5")

if i.value > 5 then
  orion.PrintDiag("i is greater than 5")
elseif i.value == 5 then
  orion.PrintDiag("i is equal to 5")
else
  orion.PrintDiag("i must be less than 5")
end
```

and

Performs the logical `and` operation on two variables or expressions.

Syntax: `operand1` **`and`** `operand2`

where the operands can be a variable or an expression.

Parameters: `and` is a binary operator, i.e. it requires a variable or expression on the left side and on the right side.

Return Value: Result of the `and` operation as follows.

Operand 1	Operand 2	Result
0 – False	0 – False	0 – False
0 – False	1 – True	0 – False
1 – True	0 – False	0 – False
1 – True	1 – True	1 – True

Example:

```
local br1 = {}
local br2 = {}

br1 = orion.GetPoint("Breaker 1 @Sub1")
br2 = orion.GetPoint("Breaker 2 @Sub1")

if br1.value and br2.value then
  orion.PrintDiag("Both breakers closed")
else
  orion.PrintDiag("At least one breaker not closed")
end
```

or

Performs the logical `or` operation on two variables or expressions.

Syntax: `operand1` **`or`** `operand2`

where the operands can be a variable or an expression.

Parameters: `or` is a binary operator, i.e. it requires a variable or expression on the left side and on the right side.

Return Value: Result of the `or` operation as follows.

Operand 1	Operand 2	Result
0 – False	0 – False	0 – False
0 – False	1 – True	1 – True
1 – True	0 – False	1 – True
1 – True	1 – True	1 – True

Example:

```
local br1
local br2

br1 = orion.GetPoint("Breaker 1 @Sub1")
br2 = orion.GetPoint("Breaker 2 @Sub1")

if br1 or br2 then
  orion.PrintDiag("At least one breaker closed")
else
  orion.PrintDiag("No breaker closed")
end
```

Math Operations

+
-
*****
/
%
^

Perform a mathematical operation.

Syntax:

```
operand1 + operand2
operand1 - operand2
operand1 * operand2
operand1 / operand2
operand1 % operand2
operand1 ^ operand2
```

where the operands can be a variable or an expression.

Parameters: +, -, *, /, %, and ^ are binary operators, i.e. each requires a variable or expression on the left side and on the right side.

Return Value: Result of the mathematical operation.

Example:

```
local i

i = 5 + 8     --i is now 13 (add)
i = i - 3     --i is now 10 (subtract)
i = 4 * i     --i is now 40 (multiply)
i = 120 / i   --i is now 3 (divide)
i = 5 % i     --i is now 2 (remainder of division)
i = i ^ 8     --i is now 256 (2 to the power of 8)
```

math.abs Returns the absolute value of the specified number.

Syntax: `math.abs(Number)`

Parameters:

`Number`	Number or expression whose absolute value will be returned.

Return Value: Absolute value of the given number.

Example:

```
local i

i = math.abs(-5) --Returns 5
```

---

math.asin
math.acos
math.atan Perform an inverse trigonometric operation.

Syntax:

```
math.asin(Number)
math.acos(Number)
math.atan(Number)
```

Parameters:

`Number`	Ratio from which angle in radians is computed.

Return Value: Angle in radians.

Example:

```
local i

i = math.asin(0) --Returns 0
```

orion.Fix

Returns the integer (fixed) portion of a number. If the number is negative, `orion.Fix` returns the negative number greater than or equal to the number.

Syntax: `orion.Fix(Number)`

Parameters:

`Number`	Number or expression for which the fixed portion of its value will be returned.

Return Value: Fixed portion of the number.

Example:

```
local i

--Positive Number
i = orion.Fix(3 * 0.9) --Result i = 2

--Negative Number
i = orion.Fix(-2.2) --Result i = -2
```

orion.Int

Returns the integer portion of a number. If the number is negative, `orion.Int` returns the negative number less than or equal to the number.

Syntax: `orion.Int(Number)`

Parameters:

`Number`	Number or expression for which the integer value of its value will be returned.

Return Value: Integer value of the number.

Example:

```
local i

--Positive Number
i = orion.Int(3 * 0.9) --Result i = 2

--Negative Number
i = orion.Int(-2.2) --Result i = -3
```

math.log
math.log10

Compute the natural logarithmic value (math.log) or base-10 logarithmic value (math.log10) for supplied number.

Syntax:
```
math.log(Number)
math.log10(Number)
```

Parameters:

Number	Number for which to compute logarithmic value.

Return Value: Natural or base-10 logarithmic value for supplied number.

Example:
```
local i
i = math.log10(100)    --Returns 2
```

---

math.pow

Returns the result of taking the base parameter to the power specified by the exponent parameter.

Syntax:
```
math.pow(Base, Exponent)
```

Parameters:

Base	Number to which the power will be applied.
Exponent	Power to apply to the base

Return Value: Result of given base taken to given power.

Example:
```
local i
i = math.pow(2, 5)    --Returns 32
```

---

math.sin
math.cos
math.tan

Perform a trigonometric operation on an angle.

Syntax:
```
math.sin(Angle)
math.cos(Angle)
math.tan(Angle)
```

Parameters:

Angle	Angle in radians.

Return Value: Ratio (sine, cosine, or tangent).

Example:
```
local i
i = math.sin(45) --Returns 0.7071
```

math.sqrt Returns the square root of the specified number.

Syntax: `math.sqrt(Number)`

Parameters: `Number` Number for which the square root is needed.

Return Value: Square root of the specified number

Example:
```
local i
i = math.sqrt(49) --Returns 7
```

---

math.rad Converts an angle from degrees to radians.

Syntax: `math.rad(Angle)`

Parameters: `Angle` The angle in degrees that is to be converted.

Return Value: The value of the angle in radians.

Example:
```
local i
i = math.rad(180) --Returns pi
```

---

math.deg Converts an angle from radians to degrees.

Syntax: `math.deg(Angle)`

Parameters: `Angle` The angle in radians that is to be converted.

Return Value: The value of the angle in degrees.

Example:
```
local i
i = math.deg(math.pi()) --Returns 180
```

math.ceil

Returns the smallest integer larger than or equal to the specified number.

Syntax: `math.ceil(Number)`

Parameters: `Number` Number for which the ceiling is needed.

Return Value: Ceiling of the specified number

Example:

```
local i

i = math.ceil(3.142) --Returns 4
```

---

math.floor

Returns the largest integer smaller than or equal to the specified number.

Syntax: `math.floor(Angle)`

Parameters: `Angle` Number for which the floor is needed.

Return Value: Floor of the specified number.

Example:

```
local i

i = math.floor(3.142) --Returns 3
```

---

math.min

Returns the minimum value among its arguments.

Syntax: `math.min(x, ...)`

Parameters: `x, ...` The numbers for which the minimum is needed.

Return Value: The minimum value of the given numbers.

Example:

```
local i

i = math.min(3, 4) --Returns 3
```

math.max Returns the maximum value among its arguments.

Syntax: `math.max(x, ...)`

Parameters: `x, ...` The numbers for which the maximum is needed.

Return Value: The maximum value of the given numbers.

Example:
```
local i

i = math.max(3, 4) --Returns 4
```

---

math.exp Returns the value of *e* to the specified power.

Syntax: `math.exp(x)`

Parameters: `x` The power to apply to *e*.

Return Value: The value of *e* to the specified power.

Example:
```
local i

i = math.exp(1) --Returns e
```

---

math.pi Returns the value of *pi*.

Syntax: `math.pi()`

Parameters: None.

Return Value: The value of *pi*.

Example:
```
local i

i = math.pi() --Returns pi
```

bit.band Performs a bitwise conjunction of two or more expressions.

Syntax: `bit.band(x1 [,x2...])`

Parameters: `x1 [,x2...]` Numbers for which a bitwise conjunction is needed.

Return Value: Bitwise conjunction of the specified numbers.

Example:
```
local i
i = bit.band(1, 3) --Returns 1
```

bit.bor Performs a bitwise disjunction of two or more expressions.

Syntax: `bit.bor(x1 [,x2...])`

Parameters: `x1 [,x2...]` Numbers for which a bitwise disjunction is needed.

Return Value: Bitwise disjunction of the specified numbers.

Example:
```
local i
i = bit.bor(1, 3) --Returns 3
```

bit.bxor Performs a bitwise exclusion of two or more expressions.

Syntax: `bit.bxor(x1 [,x2...])`

Parameters: `x1 [,x2...]` Numbers for which a bitwise exclusion is needed.

Return Value: Bitwise exclusion of the specified numbers.

Example:
```
local i
i = bit.bxor(1, 3) --Returns 2
```

String Operations

`string.byte`

Returns the character code of the characters s[i], s[i+1], ... , s[j]. The default value of i is 1; the default value of j is i.

Syntax:
```
string.byte(s [, i [, j]])
```

Parameters:
`s [, i [, j]]` The string, starting position, and ending position.

Return Value:
The character code for the characters s[i], s[i+1], ... , s[j].

Example:
```
local i

i = string.byte("A") --Returns 65
```

`string.char`

Returns a string containing the character associated with the specified character code.

Syntax:
```
string.char(charcode)
```

Parameters:
`charcode` Number for which the corresponding character is to be obtained.

Return Value:
ASCII character for supplied number.

Example:
```
local sVar

sVar = string.char(66) --Returns "B"
```

string.find

Returns the start and end position of the first occurrence of a search string within a base string. A value of true as a fourth, optional argument `Plain` turns off the pattern matching facilities.

Syntax:

```
string.find(BaseStr, SrchStr [, StartPos [, Plain]])
```

Parameters:

`BaseStr`	The string which will be searched for `SrchStr`.
`SrchStr`	The string that will be searched for in `BaseStr`.
`StartPos`	Numeric value of the character position at which to begin the search. 1 is the first character and default value.
`Plain`	Boolean value that turns off pattern matching if true. Default value is false.

Return Value: nil if no occurrence of `SrchStr` was found.

If found, character position of the first and last matching character.

Example:

```
local i

--Returns 4
i = string.find("My String", "S", 1, true)
```

`".."`

Concatenates two strings and returns the resulting new string.

Syntax:

```
String1..String2
```

Parameters:

`String1`	Base string.
`String2`	This string will be added to the end of the base string.

Return Value: The combined string.

Example:

```
local sVar

sVar = "The Point is ".."ON"
--Output to Screen would be "The point is ON"
orion.PrintDiag(sVar)
```

Length Operator.

Syntax: `# String`

Parameters: `String` Base string for which length is needed.

Return Value: Length of the specified string.

Example:
```
local i
i = # "OrionLX"    --Returns 7
```

---

string.len Returns the length of the specified string.

Syntax: `string.len(String)`

Parameters: `String` The length of this string will be returned.

Return Value: Length of the specified string.

Example:
```
local i
i = string.len("OrionLX and IO")    --Returns 14
```

string.sub

Returns the substring of the specified string at the specified start position and until the specified end position.

Syntax: `string.sub(String, Start [, End])`

Parameters:

`String`	The string from which to extract a portion.
`Start`	Number indicating the starting position in `String` at which the substring will be extracted. The first character is at position 1.
`End`	Number indicating the ending position in `String` at which the substring will be extracted. If omitted, defaults to length of `String`.

Return Value: The extracted string.

Example:

```
local sVar

sVar = string.sub("The Point is ON", 5, 12)
--Output to Screen would be "Point is"
orion.PrintDiag(sVar)

sVar = string.sub("The Point is ON", 5)
--Output to Screen would be "Point is ON"
orion.PrintDiag(sVar)
```

string.lower

Returns a copy of the specified string with all uppercase letters changed to lowercase.

Syntax: `string.lower(String)`

Parameters:

`String`	Base string to convert to lowercase.

Return Value: String with all uppercase letters changed to lowercase.

Example:

```
local sVar

sVar = string.lower("OrionLX")  --Returns "orionlx"
```

string.upper

Returns a copy of the specified string with all lowercase letters changed to uppercase.

Syntax: `string.upper(String)`

Parameters:

`String`	Base string to convert to uppercase.

Return Value: String with all lowercase letters changed to uppercase.

Example:

```
local sVar

sVar = string.upper("OrionLX")   --Returns "ORIONLX"
```

---

orion.StrComp

Compares two strings case-insensitively.

Syntax: `orion.StrComp(String1, String2)`

Parameters:

`String1`	First string to compare.
`String2`	Second string to compare.

Return Value:

-1 if `String1` is less than `String2`. Example: `"ABC" < "ABD"`

1 if `String1` is greater than `String2`. Example: `"ABD" > "ABC"`

0 if `String1` is equal to `String2`. Example: `"abc" = "abc"`

Example:

```
local i

--Returns 1
i = orion.StrComp("My String2", "My String")
```

---

orion.Trim

Returns the specified string with leading and lagging spaces removed.

Syntax: `orion.Trim(String)`

Parameters:

`String`	The string to be trimmed

Return Value: The specified string minus leading and trailing spaces.

Example:

```
local sVar

--Returns "My String"
sVar = orion.Trim(" My String ")
```

orion.InStr

Returns the position of the first occurrence of a search string within a base string.

Syntax: `orion.InStr(StartPosition, BaseString, SearchString)`

Parameters:

`StartPosition`	Numeric value of the character position at which to begin the search. 1 is the first character.
`BaseString`	The string which will be searched for `SearchString`.
`SearchString`	The string which will be searched for in `BaseString`.

Return Value: 0 if no occurrence of `SearchString` was found.

If found, character position of the first matching character.

Example:

```
local i

i = orion.InStr(1, "My String", "S") --Returns 4
```

---

orion.Join

Concatenates two strings and returns the resulting new string.

Syntax: `orion.Join(String1, String2)`

Parameters:

`String1`	Base string.
`String2`	This string will be added to the end of the base string.

Return Value: The combined string.

Example:

```
local sVar

sVar = orion.Join("The Point is ", "ON")
--Output to Screen would be "The point is ON"
orion.PrintDiag(sVar)
```

orion.Left Returns the specified numbers of leftmost characters.

Syntax: `orion.Left(String, Length)`

Parameters:

`String`	Base string.
`Length`	Number of characters that are to be returned, starting from the beginning of the string.

Return Value: String with the specified number of leftmost characters.

Example:

```
local sVar

sVar = orion.Left("OrionLX", 4)  --Returns "Orio"
```

---

orion.Mid Returns a specified portion of a string.

Syntax: `orion.Mid(String, Start[, Length])`

Parameters:

`String`	The string from which to extract a portion.
`Start`	Number indicating the starting position in `String` at which the substring will be extracted. The first character is at position 1.
`Length`	The length of the substring to be extracted. If this optional parameter is omitted, the remainder of `String` is returned.

Return Value: The extracted string.

Example:

```
local sVar

sVar = orion.Mid("The Point is ON", 5, 8)
--Output to Screen would be "Point is"
orion.PrintDiag(sVar)

sVar = orion.Mid("The Point is ON", 5)
--Output to Screen would be "Point is ON"
orion.PrintDiag(sVar)
```

orion.Right

Returns the specified numbers of rightmost characters, counting from the end of the string.

Syntax: `orion.Right(String, Length)`

Parameters:

`String`	Base string.
`Length`	Number of characters that are to be returned, counting from the last character of the string.

Return Value: String with the specified number of rightmost characters.

Example:

```
local sVar

sVar = orion.Right("OrionLX", 4)  --Returns "onLX"
```

Timer Operations

---

orion.DisableTimer Disables the specified timer.

Syntax: `orion.DisableTimer(String)`

Parameters:

`String`	Name of the timer that is to be disabled.

Return Value: None

Example:
```
orion.DisableTimer("Timer1") --Disables Timer1
```

---

orion.EnableTimer Enables the specified timer.

Syntax: `orion.EnableTimer(String [, Interval])`

Parameters:

`String`	Name of the timer that is to be enabled.
`Interval`	Optional argument to specify timer interval in milliseconds. If omitted, interval specified at time of creation will be used.

Return Value: None

Example:
```
orion.EnableTimer("Timer2") --Enables Timer2
```

---

orion.CreateTimer Creates the specified timer (can be called in LOAD function only).

Syntax: `orion.CreateTimer(String, Interval [, Enable])`

Parameters:

`String`	Name of the timer that is to be created.
`Interval`	How often the timer executes in milliseconds.
`Enable`	Optional boolean value specifying whether timer is initially enabled or disabled. If omitted, timer will be enabled.

Return Value: None

Example:
```
--Creates Timer2 that will execute every 1 sec.
orion.CreateTimer("Timer2", 1000)
```

orion.DisableTimer2 Disables the specified timer.

Syntax: `orion.DisableTimer2(Handle)`

Parameters:

`Handle`	Handle of the timer that is to be disabled.

Return Value: None

Example:
```
orion.DisableTimer2(timer1_handle) --Disables Timer1
```

---

orion.EnableTimer2 Enables the specified timer.

Syntax: `orion.EnableTimer2(Handle [, Interval])`

Parameters:

`Handle`	Handle of the timer that is to be enabled.
`Interval`	Optional argument to specify timer interval in milliseconds. If omitted, interval specified when timer was created will be used.

Return Value: None

Example:
```
orion.EnableTimer2(timer2_handle) --Enables Timer2
```

---

orion.CreateTimer2 Creates the specified timer (must be called in LOAD function only).

Syntax: `orion.CreateTimer2(Function, Object, Interval [, Enable])`

Parameters:

`Function`	Function to be called when a timer event occurs based on the specified interval.
`Object`	Table containing the function. The default is a normal, global function if this argument is `nil`.
`Interval`	How often the timer executes in milliseconds.
`Enable`	Optional boolean value specifying whether timer is initially enabled or disabled. If omitted, timer will be enabled.

Return Value:

`Handle`	Timer handle for use in `orion.DisableTimer2` and `orion.EnableTimer2` function calls.

Example:
```
--Creates timer that will execute every 1 sec.
orion.CreateTimer2(feeder1.kwh_a, feeder1, 1000)
```

Time Operations

---

orion.GetDay Extracts the day from a date/time string.

Syntax: `orion.GetDay(String)`

Parameters: `String` Date/time string with format MM/DD/YYYY HH:MM:SS.

Return Value: Returns the day.

Example:

```
local i

--Returns 23
i = orion.GetDay("11/23/2007 12:14:20")
```

---

orion.GetHour Extracts the hour from a date/time string.

Syntax: `orion.GetHour(String)`

Parameters: `String` Date/time string with format MM/DD/YYYY HH:MM:SS.

Return Value: Returns the hour.

Example:

```
local i

--Returns 12
i = orion.GetHour("11/23/2007 12:14:20")
```

---

orion.GetMinute Extracts the minute from a date/time string.

Syntax: `orion.GetMinute(String)`

Parameters: `String` Date/time string with format MM/DD/YYYY HH:MM:SS.

Return Value: Returns the minute.

Example:

```
local i

--Returns 14
i = orion.GetMinute("11/23/2007 12:14:20")
```

orion.GetMonth Extracts the month from a date/time string.

Syntax: `orion.GetMonth(String)`

Parameters: `String` Date/time string with format MM/DD/YYYY HH:MM:SS.

Return Value: Returns the month.

Example:

```
local i

--Returns 11
i = orion.GetMonth("11/23/2007 12:14:20")
```

---

orion.GetSecond Extracts the second from a date/time string.

Syntax: `orion.GetSecond (String)`

Parameters: `String` Date/time string with format MM/DD/YYYY HH:MM:SS.

Return Value: Returns the second.

Example:

```
local i

--Returns 20
i = orion.GetSecond("11/23/2007 12:14:20")
```

---

orion.GetYear Extracts the year from a date/time string.

Syntax: `orion.GetYear(String)`

Parameters: `String` Date/time string with format MM/DD/YYYY HH:MM:SS.

Return Value: Returns the year.

Example:

```
local i

--Returns 2007
i = orion.GetYear("11/23/2007 12:14:20")
```

orion.GetDOW

Extracts the day of the week from a date/time string – 0=Sunday, 1=Monday, … , 6=Saturday

Syntax:
```
orion.GetDOW(String)
```

Parameters:

`String`	Date/time string with format MM/DD/YYYY HH:MM:SS.

Return Value: Returns the day of the week.

Example:
```
local i

--Returns 5 for Friday
i = orion.GetDOW("11/23/2007 12:14:20")
```

---

orion.LocalTime

Returns the local date/time at the time of the call.

Syntax:
```
orion.LocalTime()
```

Parameters: None.

Return Value: Returns a string with the current local date/time.

Example:
```
local sVar

--Returns "11/23/2007 12:14:20"
sVar = orion.LocalTime()
```

---

orion.SystemTime

Returns the UTC date/time at the time of the call.

Syntax:
```
orion.SystemTime()
```

Parameters: None.

Return Value: Returns a string with the current UTC date/time.

Example:
```
local sVar

--Returns "11/23/2007 18:14:20"
sVar = orion.SystemTime()
```

Alarming Operations

orion.GetUnacknowledgedAlarmCount

Retrieves the current number of unacknowledged alarms on the Orion. Unless the `Return to Normal` parameter is used, this operation only returns unacknowledged alarms in an alarm state.

Syntax: `orion.GetUnacknowledgedAlarmCount(Zone, [Return to Normal])`

Parameters:

`Zone`	Optional filter. The number corresponding to an alarm zone. This parameter filters results to show only alarms from the specified zone.
`Return to Normal`	Optional boolean value. If true, returns unacknowledged alarm points that have returned to normal in addition to the alarm points in an alarm state. If false or left blank, the operation only returns unacknowledged alarms in an alarm state.

Return Value: Returns the current number of unacknowledged alarms or -1 if the information is not available.

Example:

```
local i

--Returns count of all unacknowledged alarms
i = orion.GetUnacknowledgedAlarmCount()

--Returns count of all unacknowledged alarms for zone
1 including Return to Normal alarms
i = orion.GetUnacknowledgedAlarmCount(1, true)
```

orion.AcknowledgeAllAlarms Acknowledges all unacknowledged alarms.

Syntax: `orion.AcknowledgeAllAlarms(Zone)`

Parameters:

`Zone`	Optional filter. The number corresponding to an alarm zone. This parameter only acknowledges alarms for the specified zone if the point is configured with no Alarm Acknowledge Group in the Alarm/Archive/Retentive module.

Return Value: Returns true on success or false on failure.

Example:

```
local ok

--Acknowledges all unacknowledged alarms
ok = orion.AcknowledgeAllAlarms()

--Acknowledges all unacknowledged alarms for
zone 1
ok = orion.AcknowledgeAllAlarms(1)
```

Port Operations

`orion.ResetPortCounters` Resets the current number of polls/responses for the specified port.

Syntax: `orion.ResetPortCounters(Port)`

Parameters:

`Port`	`String containing number or letter corresponding to NCD port.`

Return Value: Returns true on success or false on failure.

Example:

```
local ok

--Resets polls/responsee for port 1
ok = orion.ResetPortCounters("1")
```

Objects and Object Operations

---

orion.point Returns a point object given a point name.

Syntax: `orion.point(String)`

Parameters:

`String`	Name of the point whose object is to be obtained. Example: `Breaker Status @Feeder1`

Return Value: Returns a point object corresponding to the specified name.

Example:

```
local pt1

pt1 = orion.point("Breaker Status @Feeder1")
```

:get Returns a table of point attributes for the specified point object. These attributes include "name", "index", "value", "changes", "online" status, and associated "string", if applicable.

Syntax: `:get()`

Parameters: None.

Return Value: A table containing the current name, index, value, number of changes, and online/offline status, as well as string, if applicable, of the specified point.

Example:

```
local pt1
local i = {}

pt1 = orion.point("Breaker Status @Feeder1")

--Returns the current attributes of
--point Breaker Status @Feeder1
--i.online or i["online"] contains the
--online/offline status
i = pt1:get()
```

:set Sets the "value", "changes", "online" status, "string", "duration", "offtime", and/or "pulses" of the specified database point. This operation will also automatically generate a change notification to inform other interested modules of the point's change.

Syntax: `:set (PointAttributes)`

Parameters: `PointAttributes` table entries:

`value`	New value that the point will be set to. This value should be within the set scaling range for the point. However, setting the value below or beyond the set scaling is not restricted.
`changes`	This parameter is optional and sets the number of changes for the point.
`online`	This parameter is optional and sets the point's online status if desired. False sets the point to offline. True sets the point to online
`duration`	This parameter is optional and sets the point's control duration in milliseconds. This parameter will be used in conjunction with a Trip/Close point operation.
`offtime`	This parameter is optional and sets the point's time between pulses in milliseconds. This parameter will be used in conjunction with a Trip/Close point operation.
`pulses`	This parameter is optional and sets the point's number of pulses. This parameter will be used in conjunction with a Trip/Close point operation.
`string`	This parameter is optional and sets the point's associated text.

Return Value: None.

Example:

```
local pt1

pt1 = orion.point("Breaker Status @Feeder1")

--Sets the Breaker Status @Feeder1 point
--online and sets its value to 1
pt1:set({value=1, online=true})
```

orion.timer Returns a timer object given a timer name.

Syntax: `orion.timer(String)`

Parameters:

`String`	Name of the timer whose object is to be obtained. Example: `Timer1`

Return Value: Returns a timer object corresponding to the specified name.

Example:

```
local tmr1

tmr1 = orion.timer("Timer1")
```

:disable Disables the specified timer.

Syntax: `:disable()`

Parameters: None

Return Value: None

Example:

```
local tmr1

tmr1 = orion.timer("Timer1")

tmr1:disable() --Disables Timer1
```

:enable Enables the specified timer.

Syntax: `:enable([Interval])`

Parameters:

`Interval`	Optional argument that specifies the interval in milliseconds at which the timer executes. If omitted, the interval specified at timer creation will be used.

Return Value: None

Example:

```
local tmr1

tmr1 = orion.timer("Timer1")

tmr1:enable() --Enables Timer1
```

orion.device

Returns a device object given a device name. This object is currently only available for DNP, Modbus and SEL client (serial and network) devices.

Syntax:
```
orion.device(String)
```

Parameters:

`String`	Name of the device whose object is to be obtained. Example: "`Feeder1`"

Return Value: Returns a device object corresponding to the specified name.

Example:
```
local dev1

dev1 = orion.device("Feeder1")
```

:get

Returns a table of device attributes for the specified device object. These attributes include "online" status, "scan" status, and all device configuration parameters. The "scan" status attribute will currently only be available on active DNP, Modbus and SEL client (serial and network) ports.

Syntax:
```
:get()
```

Parameters: None.

Return Value: The current online/offline status, on-scan/off-scan status, as well as all configured custom device parameters.

Example:
```
local dev1
local i = {}

dev1 = orion.device("Feeder1")

--Returns the current attributes of
--device Feeder1
--i.online or i["online"] contains the
--online/offline status
i = dev1:get()
```

:set Sets the "online" status, “scan” status, polls/responses “counter-reset”, “points-online” status, and any other device configuration parameter that is allowed to be changed for the specified device object. This operation will currently affect only active DNP, Modbus and SEL client (serial and network) devices. Only DNP client devices support the “Event Polls” configuration parameter.

Syntax: `:set(DeviceAttributes)`

Parameters: `DeviceAttributes` table entries:

`online`	This parameter is optional and sets the device’s online status if desired. False sets the device to offline True sets the device to online
`scan`	This parameter is optional and sets the device’s scan/poll status if desired. False sets the device off-scan True sets the device on-scan
`counter-reset`	This parameter is optional and resets the device’s polls and responses counters if desired. False does nothing. True resets the device polls/responses to zero.
`points-online`	This parameter is optional and sets the device’s point online status if desired. False sets the device points to offline True sets the device points to online
`Event Polls`	This parameter is applicable to a DNP client device only and sets the device’s event poll frequency in milliseconds.

Return Value: None.

Example:

```
local dev1

dev1 = orion.device("Feeder1")

--Takes the Feeder1 device off-scan, offline,
--and sets all of the device’s points offline
dev1:set({scan=false, online=false, [“points-
online”]=false})

--Sets the Feeder1 device on-scan, online,
--and sets the device’s polls/responses to zero
dev1:set({scan=true, online=true, [“counter-
reset”]=true})
```

orion.rtu

Returns an rtu object given an rtu name. This object is currently only available for DNP server (serial and network) and Modbus server (serial and network) RTUs.

Syntax: `orion.rtu(String)`

Parameters:

`String`	Name of the rtu whose object is to be obtained. Example: "`To_EMS_DNP`"

Return Value: Returns an rtu object corresponding to the specified name.

Example:

```
local rtu1

rtu1 = orion.rtu("To_EMS_DNP")
```

:get

Returns a table of rtu attributes for the specified rtu object. These attributes include "polls", "responses", and all rtu configuration parameters.

Syntax: `:get()`

Parameters: None.

Return Value: The current polls, responses, as well as all configured custom rtu parameters.

Example:

```
local rtu1
local i = {}

rtu1 = orion.rtu("To_EMS_DNP")

--Returns the current attributes of
--rtu To_EMS_DNP
--i.polls or i["polls"] contains the
--polls counter
i = rtu1:get()
```

orion.pollgroup Returns a poll group object, given a port and poll group name.

Syntax: `orion.pollgroup(Port, String)`

Parameters:

`Port`	Number or letter corresponding to an NCD port.
`String`	Name of the poll group whose object is to be obtained. Example: "`Default Poll Group`"

Return Value: Returns a poll group object corresponding to the specified name.

Example:

```
local grp1

grp1 = orion.pollgroup("1", "Default Poll Group")
```

:get Returns a table of poll group attributes for the specified poll group object. These attributes include "name", "days", "hours", "minutes", "seconds", and "milliseconds".

Syntax: `:get()`

Parameters: None.

Return Value: The current number of days, hours, minutes, seconds, and milliseconds of the specified poll group.

Example:

```
local grp1
local i = {}

grp1 = orion.pollgroup("1", "Default Poll Group")

--Returns the current attributes of
--Default Poll Group on Port 1
--i.seconds or i["seconds"] contains the number
--of seconds
i = grp1:get()
```

:set Sets the "days", "hours", "minutes", “seconds”, and/or “milliseconds” for the specified poll group object. Currently, this operation will only affect active DNP, Modbus and SEL client (serial and network) ports.

Syntax: `:set(PollGroupAttributes)`

Parameters: `PollGroupAttributes` table entries:

`days`	Number of days for the poll group.
`hours`	Number of hours for the poll group.
`minutes`	Number of minutes for the poll group.
`seconds`	Number of seconds for the poll group.
`milliseconds`	Number of milliseconds for the poll group.

Return Value: None.

Example:

```
local grp1

grp1 = orion.pollgroup("1", "Default Poll Group")

--Sets the number of seconds for
--Default Poll Group on Port 1
--to 30
grp1:set({seconds=30})
```

Wake Operations

---

orion.wake This command reactivates the Orion display screen from rest mode. This command only applies to OrionLX-CPX and OrionLX+ models.

Syntax:
```
orion.wake(":[Display.Screen]")
```
or
```
orion.wake()
```

Parameters:

`Display Screen`	This value refers to a particular display screen by number. The wake function assumes 0 if no number is specified. Add a colon in front of the display screen number as seen in the examples.

Return Value: Returns 0 on success, and can return either 1, 2, or 3 on failure.

Example:
```
local I

--Returns "0"
i = orion.wake(":0.0")
```

Appendix B – Tools and Buttons on the Input and Output Menus

For easy configuration, numerous tools and buttons are available as shown in Figure 47 and described in the sections below. The tools and buttons described in this appendix are available for the Inputs Menu, Outputs Menu, Logic Inputs Menu, and Logic Outputs Menu. These menus are very similar, so one sample screenshot is provided for each feature.

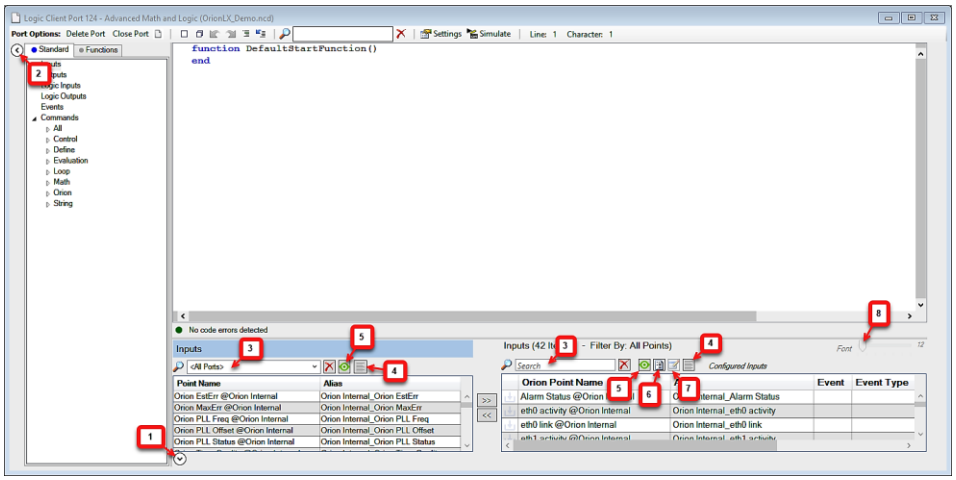


Figure 47: Tools on Input, Output, Logic Input and Logic Output Menus

1. Display/Hide Tag Name List

Clicking the button under the `Input/Outputs` lists hides the list, thus providing more usable screen space. Clicking the button displays the list again.

2. Display/Hide Menu Tree

Clicking the button to the left of the `Standard/Functions` tree hides the tree, thus providing more usable screen space. Clicking the button displays the tree again.

3. Search Point Name Lists

The `Point Name` list and the `Orion Point Name` list can be searched using free text. Enter a free text word such as `volts` in the search field. In this case, all point names containing "volts" will be filtered and displayed.

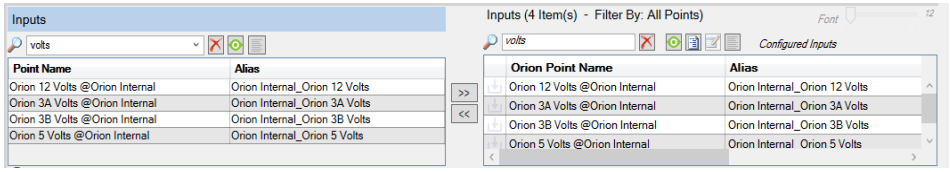


Figure 48: Filter Point Name List

The search field can be cleared by clicking on the `Clear Search` button,  to the right of the search field. Then the complete points list will be displayed again.

4. Toggle Show Point Details

To view configuration details of one or more points, highlight the point(s) in the `Point Name` list (left column) or the `Orion Point Name` list (right column) and click the `Toggle Show Details for Selected Rows` button  . An expanded window appears below the selected point and lists detailed information related to that point (Figure 49).

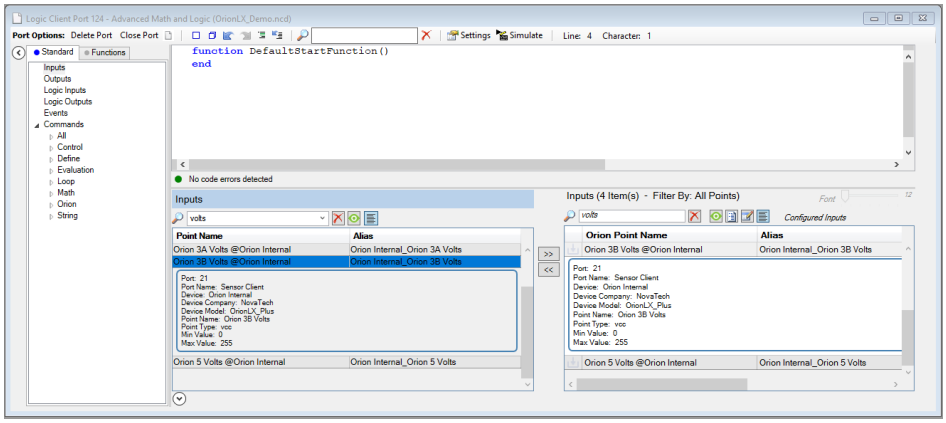


Figure 49: Show Point Details

To close the point details display, click the `Toggle Show Details for Selected Rows` button.

5. Display/Hide Alias Name

If the Add Ons option Alias Module has been configured, the `Toggle Alias Column Visibility` button is displayed above the `Orion Point Name` list in the right pane. Clicking this button will toggle the display of the `Alias` points name column.

6. View Report

The `View Report` button generates a printable report for the points which are currently displayed in the `Orion Point Name` list. This report can be printed, previewed, or saved as an HTML Web Page or as a CSV file on the PC.

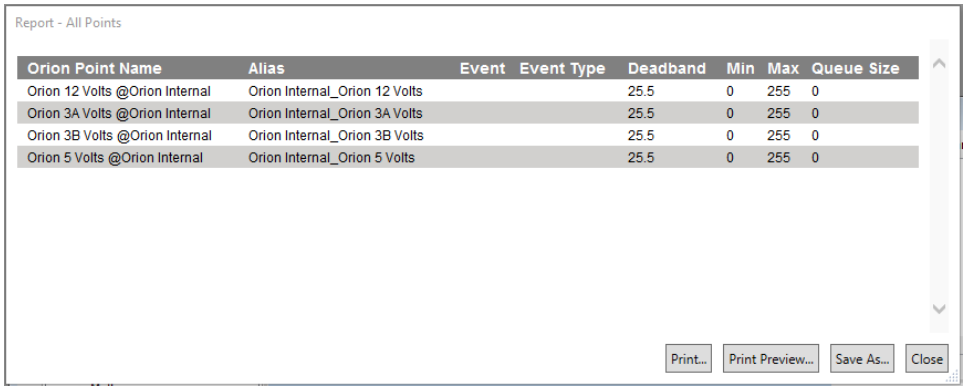


Figure 50: View Report

7. Edit Common Attributes

Note that all settings made in the `Edit Common Attributes` window can also be set directly with each individual data point.

Attributes which should be identical for multiple points configured for logic can be edited in a single window. First, highlight all points for which the attributes shall be set up identically. Then click on the `Edit Common Attributes` button.

At this point, the `Edit Common Attributes` window opens (Figure 51). Only attributes which are common to all selected points are displayed, select the attribute to be changed by clicking its checkbox, make the desired changes and click **`Apply`**. Any changes made in this window are applied to all selected points (Figure 52).

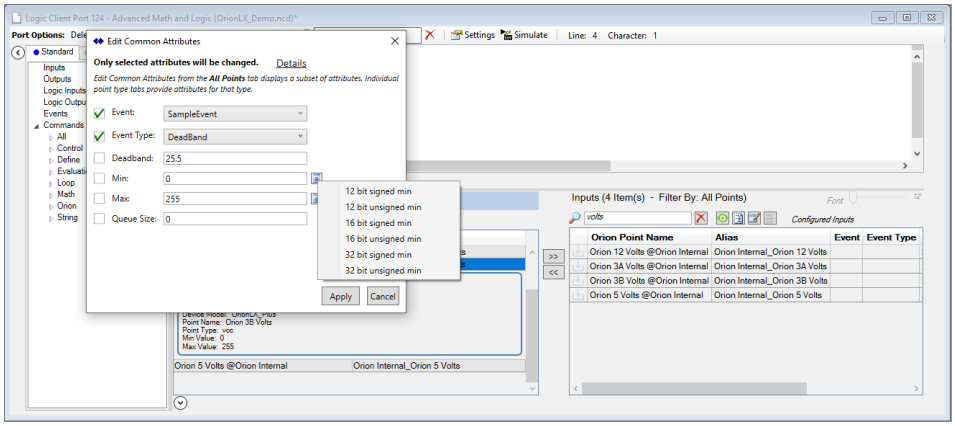


Figure 51: Edit Common Attributes

Clicking on `Details` in the `Edit Common Attributes` window lists the points for which any settings made in that window will be applied to.

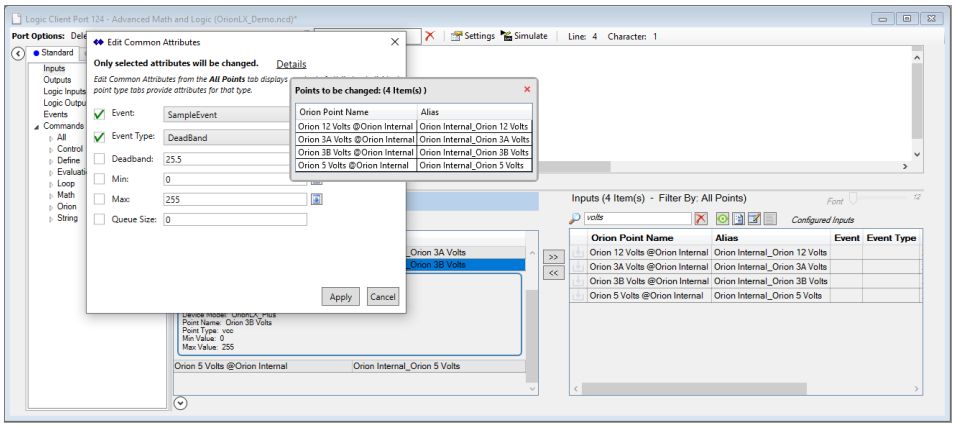


Figure 52: Edit Common Attributes - Details

8. Font Size

The font size of the `Orion Point Name` list can be adjusted by moving the slider.

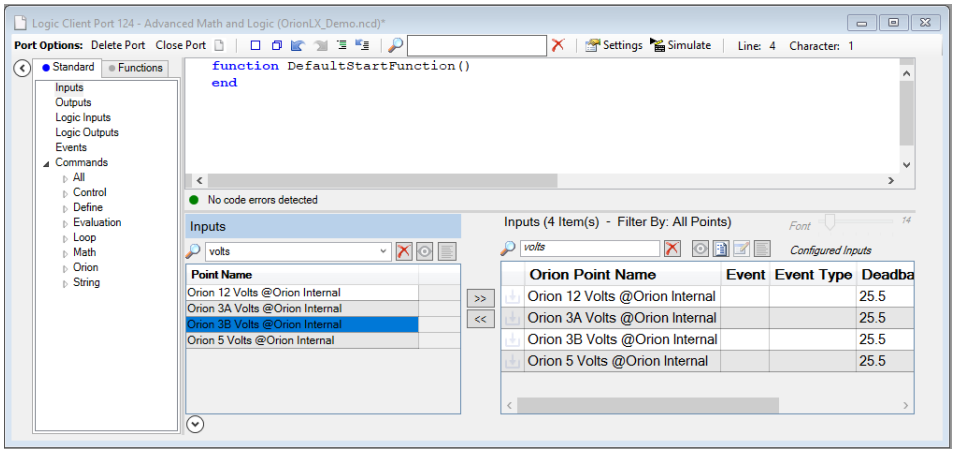


Figure 53: Font Size Slider

Appendix C – Logic in .ncd and .lua File

Commented [PR1]: Do we need this?

Below is the logic section from a .ncd file. The configuration is separated into six headings.

The `[Logic General]` heading includes a comma-separated list of .lua source files and, if applicable, the name of the function executed at startup.

The `[Logic Inputs Master]` heading contains a list of input points configured for use with the logic sub system. Each point has a name, min scaling value, max scaling value, deadband, change function, and change type. The deadband value is used for detecting data changes outside of the deadband range, while the change function is optional. The change type can be either `Refresh` for every database update or `Deadband` for every database change outside of the deadband range. These points are configured under the Logic Inputs Menu.

The `[Logic Outputs Master]` heading contains a list of output points and has identical parameters to the `[Logic Inputs Master]` heading. These points are configured under the Logic Outputs .

`[Logic Inputs Slave]` and `[Logic Outputs Slave]` are identical to `[Logic Inputs Maser]` and `[Logic Outputs Master]` respectively, with the exception that points listed in the server headings are pseudo points. Points are configured under the Inputs Menu and the Outputs Menu, respectively.

The `[Logic Timers]` heading contains a list of defined timers. Each entry has a timer name, an interval in ms, and an initial enabled state.

```
[Logic General]
SourceFiles=logic2.lua
OnLoad=MyStartFunction

[Logic Inputs Master]
Point001 @Device 1, 0, 65535,1, MyFirstChangeEvent, Refresh
Point002 @Device 1, 0, 65535,1, MyFirstChangeEvent, Refresh
Point003 @Device 1, 0, 65535,1, MySecondChangeEvent, Refresh


[Logic Outputs Master]
BinaryOutput0 @Device 1, 0, 1, 1, DisableMyTimer, Refresh
BinaryOutput1 @Device 1, 0, 1, 1, EnableMyTimer, Refresh


[Logic Inputs Slave]
Pseudo1 @Device 1, 0, 65535,1, MyThirdChangeEvent, DeadBand


[Logic Outputs Slave]


[Logic Timers]
Timer1,1,TRUE
```

Appendix D – Lua Quick Reference Libraries

The Lua Cheat Sheet

Reserved identifiers and comments

and	**break**	**do**	**else**	**elseif**	**end**	**false**	**for**	**functio**	**if**	**in**
local	**nil**	**not**	**or**	**repeat**	**return**	**then**	**true**	**until**	**while**	
`-- ...`	comment to end of line				`--[=[`	multi line comment (zero or multiple '=' are valid)				
_X is "reserved" (by convention) for constants (with **X** being any sequence of uppercase letters)					**#!**	usual Unix shebang; Lua ignores whole first line if this starts the line.				

Types (the string values are the possible results of base library function type ())

`"nil"`	`"boolean"`	`"number"`	`"string"`	`"table"`	`"function"`	`"thread"`	`"userdata"`

Note: for type boolean, **nil** and **false** count as false; everything else is true (including 0 and "").

Operators, decreasing precedence

^ (right associative, math library required)					
not		**#** (length of strings and tables)		**-** (unary)	
*****		/		**%**	
+		**-**			
.. (string concatenation, right associative)					
<	>	<=	>=	~=	==
and (stops on **false** or **nil**, returns last evaluated value)					
or (stops on **true** (not **false** or **nil**), returns last evaluated value)					

Assignment and coercion

`a = 5 b= "hi"`	simple assignment; variables are not typed and can hold different types. Local variables are lexically scoped; their scope begins after the full declaration (so that local **a** = 5).
`a, b, c = 1, 2, 3`	multiple assignments are supported
`a, b = b, a`	swap values: right hand side is evaluated before assignment takes place
`a, b = 4, 5, "6"`	excess values on right hand side ("6") are evaluated but discarded
`a, b = "there"`	for missing values on right hand side **nil** is assumed
`a = nil`	destroys **a**; its contents are eligible for garbage collection if unreferenced.
`a = z`	if **z** is not defined it is **nil**, so **nil** is assigned to **a** (destroying it)
`a = "3" + "2"`	numbers expected, strings are converted to numbers (a = 5)
`a = 3 .. 2`	strings expected, numbers are converted to strings (a = "32")

Control structures

do *block* **end**	block; introduces local scope.
if *exp* **then** *block* {**elseif** *exp* **then** *block*}	conditional execution
while *exp* **do** *block* **end**	loop as long as *exp* is true
repeat *block* **until** *exp*	exits when *exp* becomes true; *exp* is in loop scope.
for *var = start, end [, step]* **do** *block* **end**	numerical for loop; *var* is local to loop.
for *vars* **in** *iterator* **do** *block* **end**	iterator based for loop; *vars* are local to loop.
break	exits loop; must be last statement in block.

Function definition

function *name* **(** *args* **)** *body* [**return** *values*]	defines function and assigns to global variable **name**
local function *name* **(** *args* **)** *body* [**return**	defines function as local to chunk
f = function (*args* **)** *body* [**return** *values*]	anonymous function assigned to variable **f**
function ([*args*,] **...)** *body* [**return**	variable argument list, in *body* accessed as **...**
function *t.name* **(** *args* **)** *body* [**return**	shortcut for ***t.name*** = **function ...**
function *obj:name* **(** *args* **)** *body* [**return**	object function, gets *obj* as additional first argument **self**

Function call

`f (x)`	simple call, possibly returning one or more values
`f "hello"`	shortcut for **f("hello")**
`f 'goodbye'`	shortcut for **f('goodbye')**
`f [[see you soon]]`	shortcut for **f([[see you soon]])**
`f {x = 3, y = 4}`	shortcut for **f({x = 3, y = 4})**
`t.f (x)`	calling a function assigned to field **f** of table **t**
`x:move (2, -3)`	object call: shortcut for **x.move(x, 2, -3)**

The Mathematical Library [math]

Basic operations

math.abs (x)	returns the absolute value of **x**
math.mod (x, y)	returns the remainder of **x** / **y** as a rounded-down integer, for **y** ~= 0
math.floor (x)	returns **x** rounded down to the nearest integer
math.ceil (x)	returns **x** rounded up to the nearest integer
math.min (*args*)	returns the minimum value from the *args* received
math.max (*args*)	returns the maximum value from the *args* received

Exponential and logarithmic

math.sqrt (x)	returns the square root of **x**, for **x** >= 0
math.pow (x, y)	returns **x** raised to the power of **y**, i.e. **x^y**; if **x** < 0, **y** must be integer.
pow (x, y)	global function added by the math library to make operator '**^**' work
math.exp (x)	returns e (base of natural logs) raised to the power of **x**, i.e. e^**x**
math.log (x)	returns the natural logarithm of **x**, for **x** >= 0
math.log10 (x)	returns the base-10 logarithm of **x**, for **x** >= 0

Trigonometrical

math.deg (a)	converts angle **a** from radians to degrees
math.rad (a)	converts angle **a** from degrees to radians
math.pi	constant containing the value of pi
math.sin (a)	returns the sine of angle **a** (measured in radians)
math.cos (a)	returns the cosine of angle **a** (measured in radians)
math.tan (a)	returns the tangent of angle **a** (measured in radians)
math.asin (x)	returns the arc sine of **x** in radians, for **x** in [-1, 1]
math.acos (x)	returns the arc cosine of **x** in radians, for **x** in [-1, 1]
math.atan (x)	returns the arc tangent of **x** in radians
math.atan2 (y, x)	similar to **math.atan(y / x)** but with quadrant and allowing x = 0

Splitting on powers of 2

math.frexp (x)	splits **x** into normalized fraction and exponent of 2 and returns both
math.ldexp (x, y)	returns **x** * (2 ^ **y**) with **x** = normalized fraction, **y** = exponent of **2**

Pseudo-random numbers

math.random ([n [, m])	returns a pseudo-random number in range [0, 1] if no arguments given; in range [1, **n**] if **n** is given, in range [**n**, **m**] if both **n** and **m** are passed

The String Library [string]

Note: string indexes extend from 1 to #string, or from end of string if negative (index -1 refers to the last character).

Note: the string library sets a metatable for strings where the index field points to the string table. String functions can be used in object-oriented style, e.g. string.len(s) can be written s:len(); literals have to be enclosed in parentheses, e.g. ("xyz"):len().

Basic operations

string.len (s)	returns the length of string **s**, including embedded zeros (see also # operator)
string.sub (s, i [,	returns the substring of **s** from position **i** to **j** [default: -1] inclusive
string.rep (s, n)	returns a string made of **n** concatenated copies of string **s**
string.upper (s)	returns a copy of **s** converted to uppercase according to locale
string.lower (s)	returns a copy of **s** converted to lowercase according to locale

Character codes

string.byte (s [, i [, j]])	returns the platform-dependent numerical code (e.g. ASCII) of characters **s**[**i**], **s**[**i**+1], ···, **s**[**j**]. The default value for **i** is 1; the default value for **j** is **i**.
string.char (*args*)	returns a string made of the characters whose platform-dependent numerical codes are passed as *args*

Function storage

string.dump (f)	returns a binary representation of function **f()**, for later use with **loadstring()** (**f()** must be a Lua function with no upvalues)

Formatting

string.format (s [, *args*])	returns a copy of **s** where formatting directives beginning with '**%**' are replaced by the value of arguments *args*, in the given order (see *Formatting directives* below)

The IO Library

Complete I/O

io.open (fn [, m])	opens file with name **fn** in mode **m**: "r" = read [default], "w" = write", "a" = append, "r+" = update-preserve, "w+" = update-erase, "a+" = update-append (add trailing "b" for binary mode on some systems); returns a file object (a userdata with a C handle).
file:**close** ()	closes **file**
file:**read** (*formats*)	returns a value from **file** for each of the passed *formats*: "*n" = reads a number, "*a" = reads the whole **file** as a string from current position (returns "" at end of file), "*l" = reads a line (**nil** at end of file) [default], *n* = reads a string of up to *n* characters (**nil** at end of file)
file:**lines** ()	returns an iterator function for reading **file** line by line; the iterator does not close the file when finished.
file:**write** (*values*)	writes each of the *values* (strings or numbers) to **file**, with no added separators. Numbers are written as text, strings can contain binary data (in this case, **file** may need to be opened in binary mode on some systems).
file:**seek** ([p] [, of])	sets the current position in **file** relative to **p** ("set" = start of file [default], "cur" = current, "end" = end of file) adding offset
file:**flush** ()	flushes any data still held in buffers to **file**

Simple I/O

io.input ([file])	sets **file** as default input file; **file** can be either an open file object or a file name; in the latter case the file is opened for reading in text mode. Returns a file object, the current one if no **file** given; raises error on
io.output ([file])	sets **file** as default output file (the current output file is not closed); **file** can be either an open file object or a file name; in the latter case the file is opened for writing in text mode. Returns a file object, the current one if no **file** given; raises error on failure.
io.close ([file])	closes **file** (a file object) [default: closes the default output file]
io.read (*formats*)	reads from the default input file, usage as **file:read()**
io.lines ([fn])	opens the file with name **fn** for reading and returns an iterator function to read line by line; the iterator closes the file when finished. If no **fn** is given, returns an iterator reading lines from the default input file.
io.write (*values*)	writes to the default output file, usage as **file:write()**
io.flush ()	flushes any data still held in buffers to the default output file

Standard files and utility functions

io.stdin, io.stdout,	predefined file objects for stdin, stdout and stderr streams
io.popen ([prog [, mode]])	starts program **prog** in a separate process and returns a file handle that you can use to read data from (if **mode** is "r", default) or to write data to (if **mode** is "w")
io.type (x)	returns the string "file" if **x** is an open file, "closed file" if **x** is a closed file or **nil** if **x** is not a file object
io.tmpfile ()	returns a file object for a temporary file (deleted when program ends)

The Operating System Library [os]

System interaction

os.execute (cmd)	calls a system shell to execute the string **cmd** as a command; returns a system-dependent status code.
os.exit ([code])	terminates the program returning **code** [default: success]
os.getenv (var)	returns a string with the value of the environment variable **var** or **nil** if no such variable exists
os.setlocale (s [, c])	sets the locale described by string **s** for category **c**: "all", "collate", "ctype", "monetary", "numeric" or "time" [default: "all"]; returns the name of the locale or **nil** if it can't be set.
os.remove (fn)	deletes the file **fn**; in case of error returns **nil** and error description.
os.rename (of, nf)	renames file **of** to **nf** ; in case of error returns **nil** and error description.
os.tmpname ()	returns a string usable as name for a temporary file; subject to name conflicts, use **io.tmpfile()** instead.

Date/time

os.clock ()	returns an approximation of the amount in seconds of CPU time used by the program
os.time ([tt])	returns a system-dependent number representing date/time described by table **tt** [default: current]. **tt** must have fields **year**, **month**, **day**; can have fields **hour**, **min**, **sec**, **isdst** (daylight saving, boolean). On many systems the returned value is the number of seconds since a fixed point in time (the "epoch").
os.date ([fmt [, t]])	returns a table or a string describing date/time **t** (should be a value returned by **os.time()** [default: current date/time]), according to the format string **fmt** [default: date/time according to locale settings]; if **fmt** is "*t" or "!*t", returns a table with fields **year** (yyyy), **month** (1..12), **day** (1..31), **hour** (0..23), **min** (0..59), **sec** (0..61), **wday** (1..7, Sunday = 1), **yday** (1..366), **isdst** (true = daylight saving), else returns the **fmt** string with formatting directives beginning with '**%**' replaced according to *Time formatting directives* (see below). In either case a leading "!" requests UTC (Coordinated Universal Time).
os.difftime (t2, t1)	returns the difference between two values returned by **os.time()**

The Orion Library [orion]

Database Operations

orion.GetPoint (name)	returns either a specific point attribute or a table of point attributes from the real-time database **name** point.
orion.SetPoint (name, [att])	sets point attribute(s) **att** of the real-time database **name** point.
orion.AssocPoint (name, fn, [obj, typ, db])	associates the **name** point with the specified data change function **fn**.
orion.GetQueuedPoint (name)	returns a table of point attributes from the **name** point's change event. Returns **nil** if no events are available.

Logging Operations

orion.PrintDiag (s)	prints the string **s** to the Orion MMI View Communications menu or the Logic Simulator Diagnostic Output.
orion.PrintLog (s)	prints the string **s** to the Orion Event Log or the Logic Simulator Log Output.

Math Operations

orion.Fix (num)	returns the integer portion of number **num**, with negative numbers returning the negative number greater than or equal to the number.
orion.Int (num)	returns the integer portion of number **num**, with negative numbers returning the negative number less than or equal to the number.

Timer Operations

orion.DisableTimer (name)	disables the specified **name** timer.
orion.EnableTimer (name)	enables the specified **name** timer.
orion.CreateTimer (name, intvl, [enbl])	creates the **name** timer using interval **intvl** and optionally enables/disables **enbl** it.
orion.DisableTimer2 (h)	disables the timer specified by handle **h**.
orion.EnableTimer2 (h)	enables the timer specified by handle **h**.
orion.CreateTimer2 (fn, obj, intvl, [enbl])	links function **fn** with a timer using interval **intvl** and optionally enables/disables **enbl** it.

Time Operations

orion.GetDay (s)	returns the **day** from the date/time string **s**.
orion.GetHour (s)	returns the **hour** from the date/time string **s**.
orion.GetMinute (s)	returns the **minute** from the date/time string **s**.
orion.GetMonth (s)	returns the **month** from the date/time string **s**.
orion.GetSecond s)	returns the **second** from the date/time string **s**.
orion.GetYear (s)	returns the **year** from the date/time string **s**.
orion.GetDOW (s)	returns the **day of the week** using the date/time string **s**.
orion.LocalTime ()	returns a string with **local** date/time.
orion.SystemTime ()	returns a string with **UTC** date/time.

Appendix E – Additional References

Document Title	File Name
OrionLX/OrionLX+ User Manual	OrionLX_User_Manual.pdf
OrionLXm User Manual	OrionLXm_User_Manual.pdf
OrionMX User Manual	OrionMX_User_Manual.pdf
Orion I/O User Manual	OrionIO_User_Manual.pdf
Analog/Accumulator Scaling	TechNote_Scaling.pdf

Third-Party Lua References

- *Lua 5.3 Reference Manual* http://www.lua.org/manual/5.3/manual.html
- Lua database connectivity: https://keplerproject.github.io/luasql/manual.html
- Luars232 object: http://lua-users.org/lists/lua-l/2012-09/msg00554.html and https://github.com/ynezz/librs232/blob/master/doc/example.lua

Note that these references are not controlled by NovaTech and may be changed by the respective authors at any time.

Revision	Date	Changes
A	03/01/21	Initial release with logic Simulator and added NCD functionality.