Technical Document

Niagara Modbus Driver Guide

March 25, 2025

niagara4

Legal Notice

Contents

About this guide

This topic contains important information about the purpose, content, context, and intended audience for this document.

Product Documentation

This document is part of the Niagara technical documentation library. Released versions of Niagara software include a complete collection of technical information that is provided in both online help and PDF format. The information in this document is written primarily for Systems Integrators. To make the most of the information in this book, readers should have some training or previous experience with Niagara software, as well as experience working with JACE network controllers.

Document Content

This document applies to any of the Modbus drivers for any release of Niagara 4.9 and later. The audience is a knowledgeable Modbus user who is Niagara 4 certified.

Document change log

Updates, changes and additions to this guide are listed by the date the document was released.

March 25, 2025

Added two new properties "Double 64-bit Byte Order" and "Long 64-bit Byte Order" in the "Components" and "Plugins" Chapter.

- ModbusAsyncSlaveNetwork and ModbusSlaveDevice.
- ModbusSlaveNetwork and ModbusSlaveDevice.
- ModbusTcpNetwork, ModbusTcpDevice, ModbusTcpGateway and ModbusTcpGatewayDevice.
- ModbusTcpSlaveNetwork and ModbusTcpSlaveDevice.
- **New Device Properties Window** in the "Plugins" Chapter.

Added description of new Data Types available in the ModbusTcpDevice component (ModbusTcpNetwork) as of Niagara 4.14.

November 1, 2023

Added new topics “HTML5-ModbusServerPointUxManager”,and “HTML5-ModbusClientPointUxManager” to the “Plugins” chapter.

September 12, 2023

Added new topics “HTML5- ModbusAsyncDeviceManager”, “HTML5- ModbusClientDeviceManager”, “HTML5- ModbusTcpClientDeviceManager”, and “HTML5- ModbusTcpDeviceManager” to the “Plugins” chapter.

July 27, 2022

Minor updates to “Limits imposed by the Modbus licenses” topic.

February 9, 2021

Many minor edits throughout.

December 15, 2019

Document reorganized to emphasize tasks with reference material moved to the end of the document. This document has been updated for Niagara 4.9 and later.

August 25, 2010

Expanded the Preface and legal text. Added details about client Modbus Enum Bits and Numeric Bits proxy points. Added details about a client component to read function code 07 (exception status).

February 14, 2008

Added references to the *Drivers Guide*.

March 15, 2007

Completely reworked what was formerly a placeholder document providing new main chapters, numerous content changes, and summary descriptions. Also added terms.

June 24, 2005

Initial document publication.

Related documentation

These documents contain related information.

- Multiple install and startup guides, one for each remote host controller.
- *Niagara Drivers Guide*
- *Niagara Platform Guide*

Chapter 1. Getting started

Modbus is an open communications protocol originally developed in 1978 by Modicon Inc. for networking industrial PLCs (programmable logic controllers). (Modicon is now an international brand of Schneider Electric.) Since its introduction, it has gained popularity with a number of control device vendors to transfer discrete/ analog I/O and register data between control devices.

While the type of Modbus network and the devices it supports change depending on the technology (serial or TCP/IP), and the purpose of the network and devices change based on their roles as client or server (slave), the driver procedures for setting up networks, devices and points are basically the same.

Architecture

The driver provides four Modbus modules that support five Modbus network types. Three networks serve as clients, where the host (controller and station) act as a Modbus master device. In the other two networks, the controller and station serve as servers (slaves), where the station exposes Modbus data and responds to Modbus queries. All Modbus networks use the standard Framework network architecture. For more information, refer to the *Niagara Drivers Guide*.

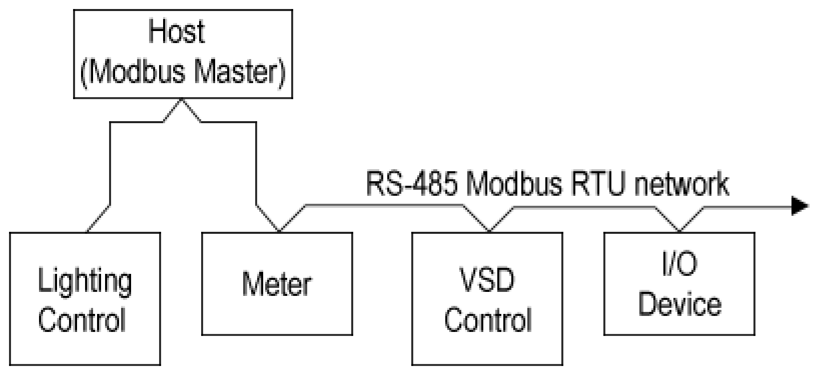
Async networks

The Modbus protocol defines the message structure and format used in communication transactions. Modbus devices communicate using a master-slave relationship in which only the master device can initiate a communication transaction. A Modbus network supports only one master device. For most integrations (modbusAsync, modbusTcp), the remote host serves as the master device. Other devices function as Modbus slaves.

Modbus provides two slave components: modbusSlave or modbusTcpSlave. Either slave station can act as a server. Usage of these components is expected to be infrequent. When used, basic Modbus principles remain the same.

Similar to other Framework integrations, the Modbus driver uses proxy points to provide monitoring and control. To help clarify station configuration, this document describes items specific to Framework components.

Figure 1. ModbusAsyncNetwork



This network requires a serial port (typically, RS-485) on the host platform, which connects to a Modbus RTU (Remote Terminal Unit) or ASCII network and functions as the Modbus Master. Lighting Control, Meter, VSD Control and I/O Device function as slaves on the same network.

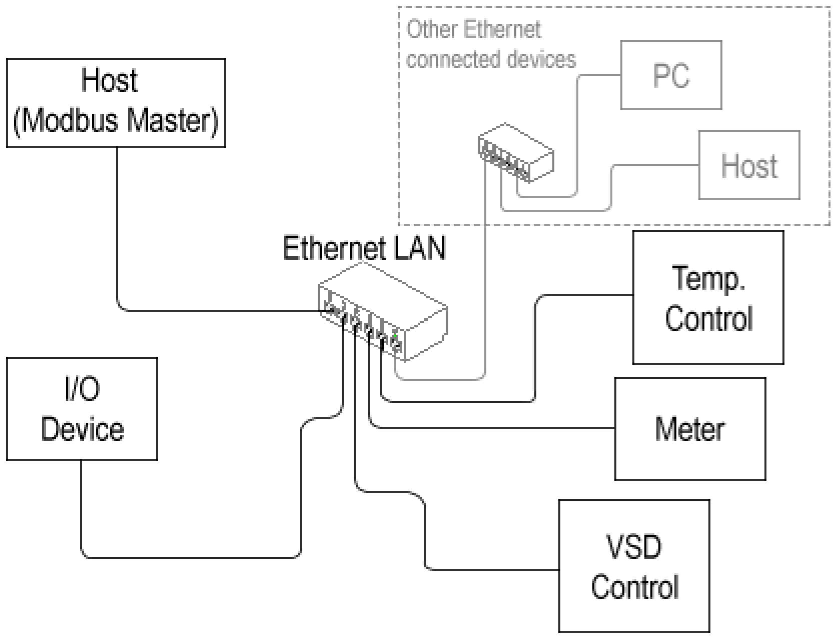
Communications rates are typically at 9600 baud, and the network transmission mode (protocol) may be either

Modbus RTU or Modbus ASCII (either one is supported). If Modbus RTU over RS-485, up to 31 slave devices may be attached—or more, if repeaters are used. The address range for Modbus devices on a serial network is from 1 to 247, however (as noted), networks are typically smaller. Depending on the number of available COM ports, a host may support multiple ModbusAsyncNetworks.

The station acts as Modbus master to all other Modbus devices on the attached COM port. A ModbusAsyncDevice represents each child device, and has a unique Modbus address (1 to 247), as well as other Modbus config data and starting addresses for Modbus data items (coils, inputs, input registers, holding registers). A Modbus network usually has many child ModbusAsyncDevices.

TCP/IP networks

Figure 2. ModbusTcpNetwork



This Modbus network automatically binds to the TCP/IP setup of the host platform's Ethernet LAN adapter. The host appears as the Modbus Master on a network of Modbus TCP slave devices. The Network connectivity protocol is Ethernet/IP.

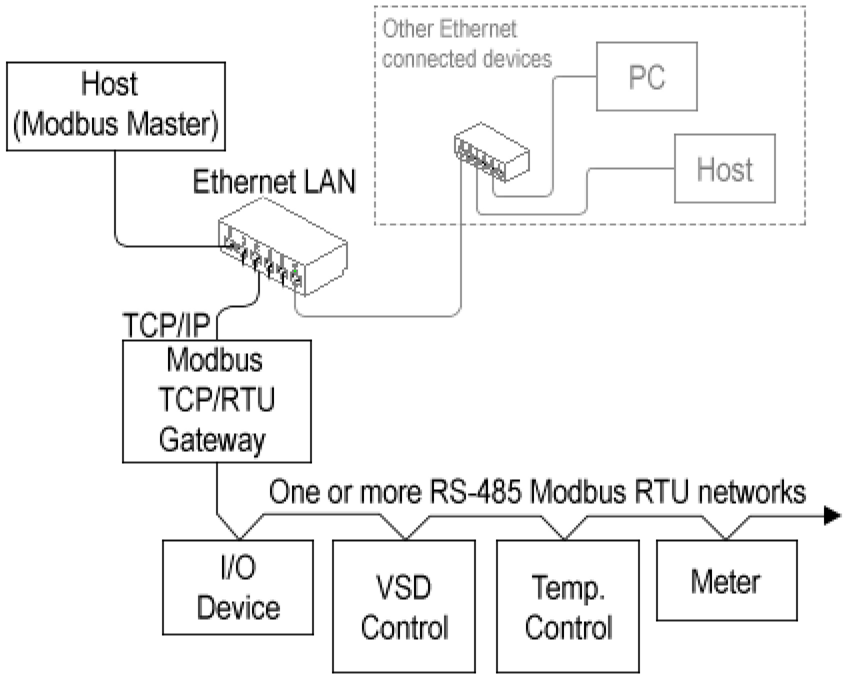
In addition to specifying the TCP software port used (typically 502), various global properties on the network's property sheet are specific to Modbus TCP. For example, you can configure the default order for float and long numeric data (overrideable within each child device).

A ModbusTcpDevice represents each child device, and has a unique IP address, as well as other Modbus config data and starting addresses for Modbus data items (coils, inputs, input registers, holding registers). There are typically many child ModbusTcpDevices.

TCP/IP gateway networks

A ModbusTCPGateway is a network-level object that also represents a particular device: a Modbus TCP-to-Serial gateway, where this device has an IP address reachable by the station. On the gateway's far side are serially-connected Modbus devices (typically Modbus RTU via RS-485). Child device objects under the gateway (network) represent each serial Modbus device (RTU or ASCII).

Figure 3. ModbusTcpGateway



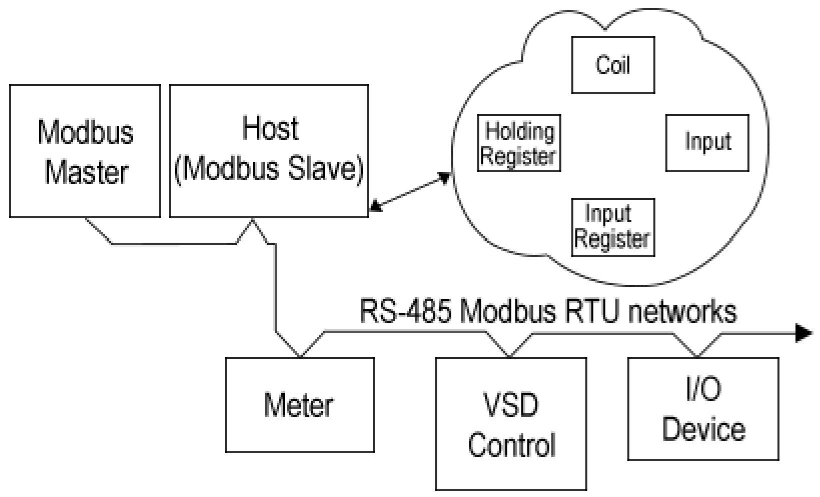
This type of network includes a Modbus TCP/Serial gateway between the Modbus Master and the serial Modbus devices.

In addition to the IP address and TCP port used by the gateway, global properties on the network's property sheet specific to Modbus configure TCP gateway devices. For example, you can configure the default order for float and long numeric data (overrideable within each child device).

The station acts as a Modbus master to the serially-connected Modbus devices on the gateway's far side. A ModbusTcpGatewayDevice represents each child device, and has a unique Modbus address (1 to 247), as well as other Modbus config data and starting addresses for Modbus data items (coils, inputs, input registers, holding registers). There are typically many child ModbusTcpGatewayDevices.

Slave networks

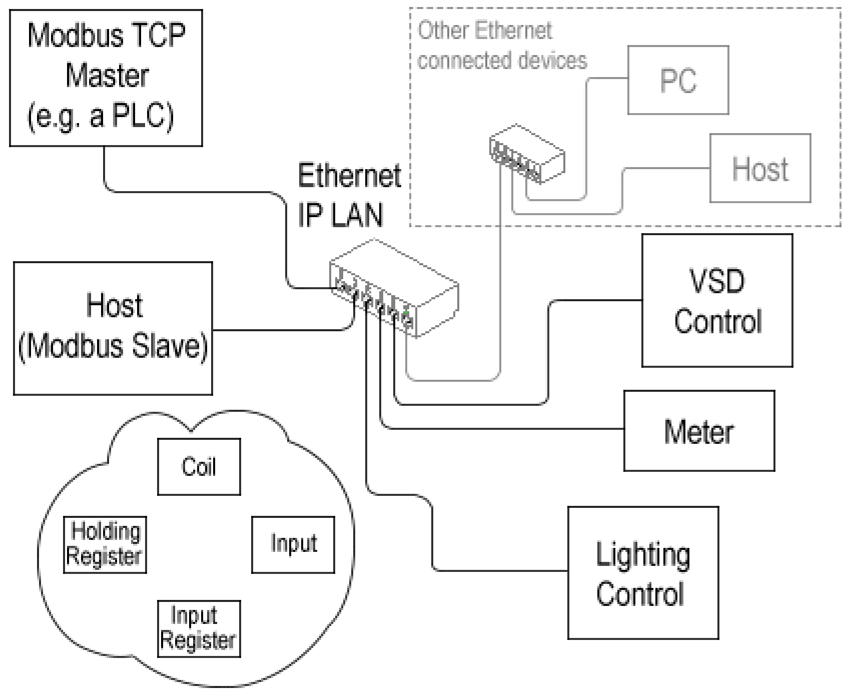
Figure 4. ModbusSlaveNetwork



This type of network requires a serial port on the host platform, which connects to a Modbus RTU or ASCII network where the station functions as a Modbus server (slave) to queries received from a serially-connected Modbus master device. In each uniquely-addressed ModbusSlaveDevice you specify the ranges for available Modbus data items (coils, inputs, input registers, holding registers). In some cases, only a single child ModbusSlaveDevice represents the station.

TCP/IP slave networks

Figure 5. ModbusTcpSlaveNetwork



This type of network automatically binds to the TCP/IP setup of the host platform's Ethernet IP LAN adapter, assumes the IP address of the station, and functions as a server. The Host (Modbus Slave) can appear as multiple devices.

The station acts as a Modbus server (slave) to queries received from a Modbus TCP master device. A ModbusTcpSlaveDevice represents each uniquely-addressed child device. You specify ranges for available Modbus data items (coils, inputs, input registers, holding registers). In some cases, only a single child ModbusTcpSlaveDevice represents the station.

Modules

To use this driver, you must have a target host (remote controller) that is licensed for the feature modbus, or a PC host, which acts as a Modbus Supervisor. The Supervisor must also be licensed for modbus).

The driver provides four Modbus modules. Each supports a different type of network.

Palette name	Network component name in the Nav tree	.jar file names in the modules folder	Function
modbusAsync	ModbusAsyncNetwork	modbuscore.jar	ProvidesModbusAsyncNetworks (serial Modbus RTU or ASCII over RS-485 or RS-232)
modbusSlave	ModbusSlaveNetwork	modbusSlave.jar	Provides ModbusSlaveNetworks (serial Modbus RTU or ASCII over RS-485 or RS-232)
modbusTcp	ModbusTcpNetwork	modbusTcp.jar	Provides ModbusTcpNetworks and/or ModbusTcpGateways (Modbus TCP via Ethernet)
modbusTcpSlave	ModbusTcpSlaveNetwork	modbusTcpSlave.jar	Provides ModbusTcpSlaveNetworks (Modbus TCP via Ethernet)

Prerequisites

To successfully install and use the Modbus driver your installation needs to meet specific requirements.

- A Niagara license for the Modbus feature(s) for each host including an optional Supervisor station running on a PC

  Other limits on devices and proxy points may exist in your license.
- A target host controller running Niagara 4.9 or later.
- Workbench running on PC.

Limits imposed by the Modbus licenses

This topic documents the limits imposed by the Modbus licenses.

RS-485 limits

`&lt;feature name="mstp"  expiration="2025-01-31"  port.limit="5"/&gt;`

`port.limit` defines the number of MS/TP trunks (RS-485 ports) that can be used. This ranges from 1 to 6, varies based on the type of host controller and depends on the license purchased.

Due to electrical considerations, the EIA-485 (or RS-485) load factor of connected MS/TP devices determines how many devices a trunk can physically support. This ranges from 31 (full load) to up to 127 (quarter load) devices.

Other device or platform limits in the license’s modbus feature also apply.

Installing the Modbus driver

This topic documents how to install the Modbus driver software in a remote host.

Prerequisites:

Niagara is installed and configured to be used as an installation tool (check box "This instance of Workbench will be used as an installation tool" enabled).

Step 1. From your PC, commission various models of remote host platforms. The .dist files are located under your Niagara install directory under a `sw` subdirectory.

Step 2. Install the modbusCore module, plus any specific modbus<type> module needed (for example, modbusAsync, modbusTcp, and so on).

Step 3. Upgrade any modules shown as out of date.

Result
The remote host is now ready for Modbus configuration in its running station.

Chapter 2. Network configuration

Some configuration tasks are shared by all the Modbus network types. Modicon introduced a variant of the Modbus protocol, Modbus TCP. This open protocol is becoming increasingly popular because it supports TCP/IP/Ethernet connectivity.

NOTE: Modicon also developed a related protocol, Modbus Plus®, which is proprietary. Compared to Modbus and Modbus TCP, the Modbus Plus protocol is not widely-used. This driver does not support Modbus Plus.

ModbusAsyncNetwork

If the host has multiple RS-485 or RS-232 ports to be used for client (master) access to Modbus networks, add one ModbusAsyncNetwork for each physical port.

Communications rates are typically at 9600 baud, and the network transmission mode (protocol) may be either Modbus RTU (Remote Terminal Unit) or Modbus ASCII (either one is supported). If Modbus RTU over RS-485, up to 31 slave devices may be attached—or more, if repeaters are used. The address range for Modbus devices on a serial network is from 1 to 247, however, networks are typically smaller. Depending on the number of available COM ports, a host may support multiple Modbus Async networks.

The station acts as the Modbus master to all other Modbus devices on the attached COM port. Each child device is represented by a ModbusAsyncDevice, and has a unique Modbus address (1 to 247), as well as other Modbus config data and starting addresses for Modbus data items (coils, inputs, input registers, holding registers). There are typically many child ModbusAsyncDevices.

ModbusTcpNetwork

Only one ModbusTcpNetwork is needed, even if the host has two Ethernet ports connected to two different (non-routed) TCP/IP LANs. The destination IP addresses of child ModbusTcpDevices automatically determine the Ethernet port.

ModbusTcpGateway

The driver supports one or more ModbusTcpGateways. Often, Modbus TCP/serial gateways are on the same LAN as other Modbus TCP devices.

Adding a Modbus network

The procedure for adding a network is the same for each network type.

Prerequisites:
The palette for the network type you are adding is open.

Step 1. Do one of the following:

- To add a single network, drag the network component from the palette to the **Station** > **Config** > **Drivers** node in the Nav tree; enter a name for the network or accept the default; and click **OK**.
- To add more than one of the same network type in a single step, double-click the station's **Drivers** container in the Nav tree; click the **New** button; select the type of network to add; enter the number of networks to add and click **OK**; name the networks or accept the defaults and click **OK**.

The Framework creates one or more network node(s) under the **Drivers** folder in the Nav tree. Initially the network status is `{fault}` and enabled as `true`. After configuring serial port and transmission mode properties, status should change to `{ok}`.

Step 2. To configure network properties, right-click the network node in the Nav tree and click **Views** >

Property Sheet

Configuring serial properties

For ModbusAsyncNetworks and the ModbusSlaveNetwork you must configure a number of properties to match the serial communications requirements of the connected Modbus devices. This includes configuring serial properties and identifying the Modbus transmission mode (RTU or ASCII).

Prerequisites:

You know how the baud rate, data bits, stop bits, parity, and flow control settings the serial network requires.

Step 1. If you have not already done so, right-click the network node in the Nav tree and select **Views** > **Property Sheet**.

The **Property Sheet** opens.

Step 2. Scroll down and expand and configure the **`Serial Port Config`** to correspond to device requirements.

Step 3. Set the **`Modbus Data Mode`** property value, either `Rtu` (default) or `Ascii`, depending on network type.

Step 4. Click the **Save** button.

Step 5. While in the property network's property sheet, review its global Modbus settings.

Configuring Ethernet properties

Ethernet properties include the gateway's IP address and port. These properties apply to the ModbusTcpNetwork, ModbusTcpGateway, and ModbusTcpSlaveNetwork components.

Prerequisites:
The network exists in the Nav tree.

Step 1. If you have not already done so, navigate to the network node, right-click it and click **Views** > **Property sheet**.
The **Property Sheet** opens.

Step 2. In the **`Ip Address`** property, enter the Modbus gateway's unique IP address, replace the `###.###.###.###` with the IP address.
Each ModbusTcpDevice you add under a ModbusTcpNetwork requires the IP address used by that device. Usually, each Modbus TCP device uses this address (only) for communications with its Modbus address (1-247) often left at 1.

Step 3. In the **`Port`** property, review the default `502` value and change this port identifier as needed.
This is the standard port used by Modbus TCP.

Step 4. While this network's property sheet is open, review its global Modbus settings.

Step 5. Click the **Save** button.

Configuring network properties

For any of the client Modbus networks (ModbusAsyncNetwork, ModbusTcpNetwork, ModbusTcpGateway), you need to configure network-level defaults for interpreting/supporting Modbus data. These defaults are on the property sheet of the network-level component.

Prerequisites:
The network exists in the Nav tree.

Step 1. If you have not already done so, navigate to the network node, right-click it and click **Views** >

Property sheet.
The **Property Sheet** opens.

Step 2. Configure **`Float Byte Order`**, **`Long Byte Order`**, **`Use Preset Multiple Register`**, and **`Use Force Multiple Coil`**, and click **Save**.

Chapter 3. Device configuration

Apart from PLCs (Programmable Logic Controllers), Modbus-capable devices provide both industrial and commercial applications, such as electric-demand meters and lighting controllers, among many others. The configuration requirements for these devices vary greatly because the Modbus protocol does not specify which specific function codes are necessary in a device. The data type and format of register-held data are left up to the vendor. And, quite commonly, different byte-order storage schemes are used for storing 32-bit data types, such as float and long (integer).

The driver's client networks (ModbusAsyncNetwork, ModbusTCPNetwork, and ModbusTCPGateway) provide four configuration properties, set at the network-level, that act as global Modbus defaults for all devices on the associated network. If needed, any (or all) of these settings can be overridden at the device-level.

Modbus messages

Modbus communication is built around messages. Each message has the same structure. A master in the network initiates a conversation with a message, which is known as a query. To any specifically addressed query, the master expects a response from the slave. A slave never initiates a transaction (sends an unsolicited message). This query-response cycle is the basis for all communication on a Modbus network. It is always the master that initiates the query, and the slave that responds.

Device address

Each message has the same structure, which is independent of the type of network. On a plain serial network the message structure is the same as the structure transmitted over TCP/IP (Ethernet). Messages consist of four parts: device address, function code, data and error check.

The address field in the query from a master defines which slave device should respond to the message. All other network nodes ignore the message if it is not addressed to them. The address field in the response from the slave contains the slave device address, which confirms to the master that the slave is replying to the query.

A device address of zero (0) indicates a broadcast message, which the master sends to all slaves and does not require a response from any slave.

Function code

Following each address field in a query is a function code, which identifies the type of information requested by the master. Function codes are also used for verification in slave responses back to the master device. Example codes are READ COIL STATUS and READ HOLDING REGISTERS.

The Modbus protocol defines 24 function codes. Few devices support all function codes. A vendor's documentation for a Modbus device should state which function codes are supported. The following table shows the function codes supported by this Modbus driver.

Table 1. Driver-supported Modbus function codes

Code	Function Name	Operation in a controller master
01	READ COIL STATUS	Provide normal data polling of all read-only and writable proxy points.
02	READ INPUT STATUS	
03	READ HOLDING REGISTERS	
04	READ INPUT REGISTERS	
05	FORCE SINGLE COIL	Manage change of values at inputs and/or invoked actions of writable proxy points or client preset components.
06	PRESET SINGLE REGISTER	
07	READ EXCEPTION STATUS	Poll device for exception status, output values and bits set.
15	FORCE MULTIPLE COILS	Manage change of values at inputs and/or invoked actions of

Code	Function Name	Operation in a controller master
16	PRESET MULTIPLE REGISTERS	writable proxy points or client preset components.
20	READ FILE RECORD	Use ModbusClientStringRecord to read and write file records in which data are converted to ASCII characters and displayed as a string.
21	WRITE FILE RECORD	

If the slave was able to perform the requested function, it returns an exact copy of the function code originally sent by the master. If the slave was unable to perform the requested function, it returns an exception response.

Data

This message field describes the particulars for the function code. For example, it may contain a register address (and a range) to read, or a coil address to write. Data fields return the information requested by the master.

Error check

This field confirms the integrity of the message as received from the master. If the slave detects an error in a query, the slave ignores the query and waits for the next query addressed to it.

In the response from the slave, this field confirms the integrity of the message as received from the slave. If the master detects an error, the master ignores the response.

The error-checking method depends on the Modbus transmission mode. Modbus RTU, the most prevalent mode, uses a CRC (cyclical redundancy checksum) method. The Modbus TCP/IP protocol has a similar message format for query and response messages. However, Modbus TCP/IP is freed from error check routines. Instead, it uses the error-checking mechanisms built into the lower-level TCP/IP and link layers (that is, Ethernet).

Modbus query and response example

The following is an example query and response message pair from a ModbusAsyncNetwork to a Modbus RTU (serial) device:

- Query

  `020300030004B43A`

  For device `02`, function code `03`, starting address `0003`, number of registers `04`, error checksum `B43A`
- Response

  `0203080051 0052003C003CA387`

  From device `02`, function code `03`, number bytes returned `08`, data (`00510052003C003C`), error checksum `A387`

Modbus data

The Modbus protocol supports a wide variety of data. The Modbus driver brings these data into the common object model in a manner that simplifies the sharing of values.

In Modbus nomenclature, the term data refers to the status of coils and inputs, which can be on or off. In the common object model these equate to two states: active or inactive (true or false). The Modbus protocol does not dictate how to format input and holding registers in terms of which data type and numerical encoding to use. A Modbus device vendor can choose any data type and configure the device with one or more consecutive 16-bit registers.

NOTE: The device vendor should document the data type used for each input and holding register. It is important to configure device proxy points accordingly.

Numeric data types

The Modbus protocol provides four numeric data types for input and holding registers. The driver supports these registers using standard Framework Boolean, Numeric and Enum data types.

- **`Integer`** represents an unsigned 16-bit numeral using a single register that ranges from 0 to 65,535. In some systems (or devices), the term "word" refers to this unsigned, 16-bit integer value.

  This is, perhaps, the most popular data type. It is the driver's data type for an unsigned, 16-bit integer value, and is the default data type for many newly-created Modbus proxy points.
- **`Float`** refers to a floating-point computation using a 32-bit single-precision scaling base, sometimes called "real." Very small and large numbers are possible. This data type requires two consecutive registers. In addition, the driver supports two byte-order schemes for float values (3-2-1-0 or 1-0-3-2).

  Proxy points automatically allocate two consecutive registers for each data item whenever you specify a float or long data type. Keep this in mind when specifying the number of points within any range of Modbus registers.
- **`Long`** represents a signed, 32-bit integer ranging from -2,147,483,648 to 2,147,483,647. This data type requires two consecutive registers and the same byte-order scheme as float data.
- **`Signed Integer`** represents a signed, 16-bit integer ranging from -32,768 to 32,767. This integer is sometimes called "short."

You define the **`Data Type`** for each NumericPoint or NumericWritable in a proxy point's proxy extension. You may configure the byte order scheme for two-register numeric values (float and long) at the device level, or globally at the network level. When dealing with 32-bit values, such as long or float values, configuration includes identifying the byte-order scheme for the two consecutive registers as processed in the device.

Bit proxy point extensions

Some vendors use a device's holding or input registers to represent a number of Boolean statuses (on and off states) in a single register with each bit indicating (mapping to) a separate status. The Modbus driver provides a special Register Bit proxy point extension to read and write to such registers, accessing each bit independently for each proxy point.

String proxy point extensions

Although not common, a Modbus device may use a number of consecutive holding registers to represent a string of alphanumeric (ASCII encoded) characters. The Modbus driver provides a String proxy point extension to read the character strings used by these string proxy points.

Driver data

With the exception of the String control point with the Modbus String proxy point extension, the driver represents all data on proxy point inputs and outputs using these data types:

- **`Boolean`** — BooleanPoint and BooleanWritable represent the two-state data identified as Modbus coils and inputs. Less frequently, Boolean data are mapped into the bits of a single input or holding register. All Modbus Boolean proxy points provide facets, which you can individually edit to match the vendor's documented state descriptions, such as on and off, enabled and disabled.
- **`Numeric`** — NumericPoint and NumericWritable represent numeric data in holding registers or input registers, whether a Modbus Float proxy point's selected **`Data Type`** is integer, long, float, or signed integer. All Modbus numeric proxy points provide facets, which you can individually edit to define minimum and maximum values, precision, and data units.
- **`Enum`** — (Modbus 3.5.26 or higher) EnumPoint and EnumWritable represent data in holding registers or input registers using the integer (ordinal) value that results from a range of consecutive bits, which are specified by a starting bit and number of bits.

Rounding values

The Modbus driver's NumericWritable proxy points, which write values to holding registers, round (and possibly restrict) input values before any write. Rounding depends on the proxy point's selected **`Data Type`**:

- **`Integer`** data types round input values up or down to the nearest whole number, and restrict all values to the range from 0 to 65,535.

- **`Long`** data types round input values up or down to the nearest whole number. The range (-2,147,483,648 to 2,147,483,647) matches the driver's value range.
- **`Signed Integer`** data types round input values up or down to the nearest whole number and restrict all values to the range from -32,768 to 32,767.
- **`Float`** data types do not round. Input values remain as they are.

Adding a device

This procedure documents how to add a single device or a group of devices to any type of Modbus network. For general device manager information, refer to the *Drivers Guide*.

Prerequisites:
You have address information for each device as well as its data configuration (coils, inputs, input registers, and holding registers. The device vendor's documentation is available.

Step 1. In the Nav tree or in the **Driver Manager** view, double-click the client network.

The appropriate device manager opens (**Modbus Async Device Manager**, **Modbus Tcp Device Manager**, or **Modbus Tcp Gateway Device Manager**).

NOTE: Device types are specific to a particular parent network type—for example, you cannot copy a ModbusSlaveDevice under a ModbusTcpSlaveNetwork, or a ModbusTcpSlaveDeviceunder a ModbusSlaveNetwork. This is not a problem when working in the device manager for either of the slave networks, as the New function (to add devices) automatically selects the proper child device component.

Step 2. Click the **New** button.

The **New** window opens. The driver preselects the **`Type to Add`** based on the network type (ModbusAsyncDevice, ModbusTcpDevice, or ModbusTcpGatewayDevice).

Step 3. Enter for number to add: `1` (or more, if multiple) and click **OK**.

A second **New** window opens with additional properties to configure the Modbus. The default values should be sufficient at least to start with, except that:

- Async, TCP/IP Gateway and slave devices require a unique Modbus address between 1-247. This address must be unique from any other physical device on that network.
- TCP/IP devices require a unique IP address (the device's Modbus address can remain at 1).
- TCP/IP devices that use a TCP port other than the standard 502 port require port definition.

You might leave these properties at their defaults, particularly if different devices use the same settings—in which case you could adjust them (globally) at the network level.

Step 4. Enter the device address in the **`Starting Address`** property, and click **OK**.

You should see the device(s) listed in the Modbus device manager view with a status of `{ok}` and enabled set to `true`.

If a device shows `{down}`, check the configuration of the network and/or the device addresses.

Step 5. To confirm or modify device properties, double-click the device row in the device manager view.

Step 6. After making any changes, click **Save**.

Step 7. Right-click the device row in the device manager and click **Actions** > **Ping**
Receiving any response from the device, including an exception response, is considered proof of communication. If the system reported that the device was previously `{down}`, any ping response is good news.

The default ping address for a client Modbus device is for the first (40001 Modbus) holding register value (integer). Often this address works well without an exception response. However, it is recommended that you confirm this ping address, and, if necessary, change in the device's property sheet.

Duplicating devices

Configuring multiple devices at once is useful in cases where all devices in the range require a different set of proxy points, however, in cases where you have like devices, you might create a single device first, configure its proxy points and other components, and then duplicate it as many times as needed. You can change the device address of each duplicate to a unique number.

Prerequisites:
You already added and configured device properties for a single device.

Each proxy point requires a unique address. Configuring a base address in a device provides a way to customize proxy point addresses in one step.

Step 1. Right-click the device in the Nav tree and click `Duplicate`.
The **Name** window opens.

Step 2. Define a name for the device and click **OK**.
Each proxy point requires a unique address. Configuring a base address in a new device provides a way to customize proxy point addresses in one step.

Step 3. To locate the base address properties, double-click the new device node in the Nav tree and expand the base address properties (**`Input Register Base Address`**, **`Holding Register Base Address`**, **`Coil Status Base Address`**, and **`Input Status Base Address`**).

All base address properties of a device default to hexadecimal zero (hex: 0). However, you can use them as an engineering method (along with multiple device objects) to quickly configure unique proxy point addresses. This works when a Modbus device has data partitioned into multiple areas with repeating address patterns. This way, the same device (and child proxy points) can be replicated, and the only address changes made in the base address.

Step 4. Select **`Hex`** or `Decimal` for the **`Address Format`**.
In this application, you cannot use the option, `Modbus` for the base address and child proxy point addresses.

Step 5. Define the base **`Address`**.
For example, if a device's **`Holding Register Base Address`** is set to `Decimal`, and its **`Address`** is set to `100`, each child Modbus proxy point that has a holding register **`Address Format`** of `Decimal`, and an **`Address`** of `13` is effectively addressed as Decimal, 113 (Absolute Address).

Step 6. When you are finished configuring properties, click **Save**.

Creating proxy points

As with device objects in other drivers, each client Modbus device has a **Points** extension that serves as the container for proxy points. The default view for any **Points** extension is a **Point Manager**. You use it to add Modbus client and slave proxy points under the appropriate Modbus device.

Prerequisites:
If you are creating server (slave) proxy points, you already created the registers for the points.

NOTE: Unlike the point managers in many other drivers, the Modbus point managers do not offer a learn mode with a **Discover** button, **Discovered**, and **Database** panes. The simplicity of the Modbus protocol renders this function unnecessary. Instead, you use the **New** button to create proxy points, referring to the vendor's documentation for the addresses of data items in each Modbus device.

Step 1. Double-click the network node in the Nav tree, under the Exts column, double-click the Points icon ( ).

The **Point Manager** view opens.

Step 2. (Optional) To help organize your points, click the **New Folder** button and create a new points

folder, giving it a short name that works for your application, then double-click it to open its **Point Manager**.
You can repeat this step to make multiple points folders or simply skip this step to make all proxy points in the root table.

Each points folders has its own point manager view.

Step 3. To create a point, do one of the following:

- Drag or copy the appropriate point form the palette to the device in the Nav tree.
- Click the **New** button and select the type of point(s) to add (**`Type To Add`**).

  Typically, this is the quickest way to add proxy points, because you can specify a number of points if they can be consecutively addressed.

The **New** points window opens.

Step 4. Select the type of point to add.

Step 5. To configure consecutive register numbers, enter the total number of points to add in **`Number To Add`**.
The screen capture shows eight added points. When you specify more than 1 point, additional points are automatically assigned consecutive addresses—relative to the **`Starting Address`** you specify for the first point.

Step 6. For the **`Starting Address`**, as a general rule, choose `Modbus` addressing for **`Address Format`**. This lets you enter target data addresses directly from the device's documentation, without having to subtract 40001, for example, or perform other mental math.

For read-only client points, using the Modbus address format frees you from having to set the register type property (**`Reg Type`**), as the Modbus address automatically sets this property going by the leading numeral of the full Modbus address (3 for input registers, and 4 for holding registers).

NOTE: When entering a Modbus formatted address for a coil, the driver ignores leading zeros. For example, the Modbus address 00109 is the same as entering Modbus 109. Unlike Decimal or Hex address formats (zero-based formats), the Modbus address format is one-based, meaning that a coil addressed as Modbus 109 has a Decimal address of 108, and a Hex address of 6D.

Step 7. If the point(s) is/are numeric select the **`Data Type`**, then click **OK**.

This opens another **New** window.

Step 8. Name the point(s), enter data addresses as well as enter other information, such as point facets and conversion, and click **OK**.
The driver adds the proxy point(s) to the **Points** extension (or to the current points folder) where each shows as a row in the point manager. If addressed correctly, each point should report a status of `{ok}` with a polled value displayed.

Step 9. If a point reports a `{fault}` status, check its ProxyExt **`Fault Cause`** property value, which typically includes a Modbus exception code string, such as `Read fault: illegal data address`.

Step10. Continue to add proxy points as needed under the **Points** extension of each client Modbus device.

Step11. To edit a point, double-click its row in the manager.

Result
If programming online and the device shows a status of `{ok}`, you can get statuses and values back immediately to help determine if point configuration is correct. Modbus server proxy points must fall within the defined address register ranges of the parent server (slave) Modbus device, otherwise they will retain a fault status.

Example

A Modbus energy meter device has a number of TOU (Time of Use) parameters, including several for TOU Tariff Change Time configuration. For the eight possible tariff periods (1 - 8), there is a start hour (0 - 23) and start quarter of an hour (0 - 3). In the meter device, a single 16-bit holding register holds the setup of all eight tariff periods, mapped from highest bit (15) to lowest bit (0) as follows:

- Bits 8:15 = 0 - 7 (corresponding to tariff #1 - #8)
- Bits 2:7 = 0 - 23 (tariff start hour)
- Bits 0:1 = 0 - 3 (tariff start quarter of an hour)

Three separate proxy points can be created, using either NumericBitsWritables or EnumBitsWritables (depending on user interface requirements), to provide read/write access to these bit-mapped values. All three points will specify the same (register) Data Address, but have different Beginning Bit values and Number Of Bits values.

Configuring a device for polling

Data polling in a client device may be improved by configuring the use a single message to poll consecutively addressed values. This reduces network messaging traffic. After adding all proxy points under a device, configure the device for polling. Typically, this improves polling response due to fewer messages to get the same amount of data.

Prerequisites:
You have created the device and added proxy points. Device polling should be configured until after proxy points are created, and typically already receiving values from (individual) point polling.

Step 1. Double-click the **ModbusAsyncDevice**, **ModbusTcpDevice**, or **ModbusTcpGatewayDevice** in the Nav tree and expand the **`Device Poll Config`** property.
The **`Modbus Tcp Device`** property sheet opens.

Step 2. Configure the **`Start Address`** and **`Consecutive Points To Poll`**.

Step 3. To organize points, add child **DevicePollConfigEntry** objects manually in this container or (optionally) use the container's right-click action: `Learn Optimum Device Poll Config`.

Chapter 4. Client (master) operations

The Modbus driver supports three types of client devices: ModbusAsyncDevice, ModbusTcpDevice, and ModbusTcpGatewayDevice. Each client device component represents a remote Modbus slave. The host station on a Modbus master network regularly polls these slave devices with requests, which provide Modbus data. Remote client devices listen for these Modbus queries from the master (host station) and send responses. Each type of client supports proxy points. Data exchange occurs with both writable and read-only proxy points, client preset objects, and (if needed) reads and writes to file records, for string data.

Each type of client device is specific to a particular type of parent network. You cannot drag a ModbusAsyncDevice from a palette to a ModbusTcpGateway or a ModbusTcpGatewayDevice to a ModbusTcpNetwork. This is not a problem when working in the device manager for any of the three client networks, as the **New** window (used to add devices) automatically selects the proper child device component.

The three types of Modbus client devices are similar in that each has a single frozen **Points** device extension managed by the default **Modbus Client Point Manager** view. Each **Points** device extension supports the same type of proxy points, as well as preset and file record objects.

In addition to common **Points** slots, all three types of Modbus client devices provide similar properties. This includes overrides of network level Modbus Config settings, ping address setup for the parent network's Monitor ping, device base address configurations for Modbus data items, and slots for configuring device-level polling.

Configuring a client device for polling

A device-level polling of points permits a single Modbus query message to retrieve a number of consecutive data values. Device-level polling may help overall polling efficiency by reducing the number of polls necessary at the point-level. The table of device poll configuration entries under the frozen slots ModbusAsyncDevice and ModbusTcpGatewayDevice specify the polling properties.

Prerequisites:
All needed proxy points exist under a device.

NOTE: In a few cases, a device-level poll did not improve polling efficiency. The target Modbus device took more time to assemble a long data response than it did to handle a number of separate responses (no device poll, point-level polling only) for the equivalent data. While not typical, you should be aware that Modbus devices vary in performance.

Step 1. In the Nav side bar, expand the Modbus device so you see its **Device Poll Config** slot (or, open the property sheet of the Modbus device to see this same slot listed with other properties and slots).

Step 2. To clear any existing polling entries, click **Actions** > **Clear**.

Step 3. To configure polling in a single step right-click `Device Poll Config`, and select **Actions** > **Learn Optimum Device Poll Config**.

The learn algorithm looks for any consecutively addressed Modbus proxy points, and creates a DevicePollConfigEntry when it finds two or more consecutively addressed points. If you have small gaps between consecutively addressed Modbus proxy points, you may want to manually adjust the created DevicePollConfigEntries to poll over the small gaps. You can always create, configure, and remove DevicePollConfigEntries until you find the most efficient device-polling scheme. This action-learn is the most common method for setting up polling properties. It lets you replace existing device poll entries (start over) or append to the existing device polls.

Step 4. To manually edit polling properties, duplicate or copy them and edit the copies.

Adding client presets

Presets allow writing preset values to the target addressed data items upon right-click action (command). If needed, you can link the Write action of any preset container to other control logic. For example, you can link the Trigger slot of a TriggerSchedule into a ModbusClientPresetRegisters container component for periodic writes to its child preset registers based upon some repeating schedule.

In cases where you want to write preset values to specified coils and/or holding registers in a Modbus device using a linkable Write action, you can add special Preset components under the device. These are not actually proxy points—you need to copy them from the modbusAsync or modbusTcp palette. In rare cases, you may also wish to read/write string data from Modbus files in a device. The palettes also have a component especially for this application, which you can also copy under the Modbus device.

- The Modbus Client Preset Coils preset contains a single ModbusClientPresetCoil to write to one coil—you can add additional (consecutive) coils using a built-in action
- The Modbus Client Preset Registers preset contains a single ModbusClientPresetRegister to write to one holding register—you can add additional (consecutive) registers using a built-in action

NOTE: Modbus client preset components are not proxy points. There is no ProxyExt as data is not polled and read from the device—only written to it. You do not see them in any **Modbus Client Point Manager** view if you copy them under a client device's **Points** extension. Presets exist for both Modbus coils and holding registers. These are the only two Modbus data items to which a Modbus master may write.

Instead of adding presets as described in this procedure, you can copy and paste already configured presets from another client Modbus device, if appropriate.

Step 1. Open the modbusAsyncor modbusTcp palette in the Palette side bar.

Step 2. In the Nav side bar, expand the client Modbus network to show the Modbus device of interest.

Step 3. Do one of the following:

- To configure coils or holding registers, drag or copy one of the preset container components to the Modbus device in the Nav side bar (or, to the property sheet view of that device, if open).
- To configure presets for both coils and holding registers, copy the entire Presets folder from the palette.

Since preset components are not proxy points, you should locate them elsewhere under the device.

NOTE: If using multiple preset containers under a client Modbus device, be careful not to overlap preset addresses. In other words, any specific preset address should be in only one preset container.

Step 4. In the **Name** window, accept the default name or enter an alternate name, and click **OK**.

The folder is added under the device. By default it contains two preset containers, each with one preset entry—one for a preset coil, one for a preset holding register. Either preset container can be deleted (if not needed), or duplicated, as well as have additional preset entries added.

Step 5. In any preset container, configure the **`Starting Address`** and other property values, and in its child preset entries (coil or register types), enter the actual preset values.

NOTE: You do not have to copy the entire Presets folder from the palette—this is just the easiest way to add both preset containers, each with a single preset entry child. You can locate preset containers anywhere under the Modbus device. However, be aware that if you copy these components under the **Points** container, they are not visible in any **Modbus Client Point Manager** view.

Step 6. To add a preset to an existing Coil or Preset Register container, right-click the container and click **Actions** > **Add Preset Coil Value** or **Actions** > **Add Preset Register Value**.

The **Add Preset ...** window opens.

By default, added client preset components are appended to the bottom of the slot order.

Step 7. To change the address order of the child presets relative to the absolute address in the preset container, right-click on a preset container and select `Reorder`.

Adding file records

A Modbus client string record provides for the reading and writing of Modbus file records (client side support for Modbus function codes 20 and 21). The input and output is a string converted to and from a byte array. Writing occurs when the linkable write action is fired, and reading occurs when the linkable read action is fired.

Step 1. Open the modbusAsync or modbusTcp palette in the Palette side bar, if not already opened.

Step 2. In the Nav side bar, expand the client Modbus network to show the Modbus device of interest.

Step 3. From the palette, drag the **`Modbus File Records`** folder onto the Modbus device in the Nav side bar (or, into the property sheet view of that device, if open).

Step 4. In the popup **Name** window, accept the default **`Modbus File Records`** name or enter an alternate name, and click **OK**.

The folder is added under the device. By default it contains a single ModbusClientStringRecord component. You can duplicate it if multiple file record objects are needed.

Step 5. Double-click the added component to open its property sheet, and enter appropriate configuration values per the Modbus device vendor's documentation.

NOTE: You do not have to copy the entire Modbus File Records folder from the palette—this is simply the easiest way to add the needed component with a descriptive parent folder. You can locate ModbusClientStringRecord components anywhere under the Modbus device. However, be aware if you copy these components under the Points container, they are not visible in any **`Modbus Client Point Manager`** view.

Chapter 5. Server (slave) configuration

The station acts as a Modbus slave (server) to queries received from a connected Modbus master device. In each uniquely-addressed ModbusSlaveDevice, you specify the ranges for available Modbus data items (coils, inputs, input registers, and holding registers). In some cases, only a single child ModbusSlaveDevice may exist to represent the station.

Modbus server (slave) devices include: ModbusSlaveDeviceand ModbusTcpSlaveDevice. Both represent the station as a virtual Modbus slave, that is, the station listens for Modbus queries from a remote Modbus master, and sends responses.

Both Modbus server devices are similar, having a single frozen **Points** device extension, with the default Modbus Server Point Manager view. The same type of Modbus server proxy points are used under **Points** device extensions.

In addition to common device slots, both types of Modbus server devices have similar properties for Modbus configuration. This includes overrides of network level device data settings and register range configurations for Modbus data items.

Modbus registers

A Modbus device holds transient (real-time) data and often persistent (configuration) data in addressable registers. The term “register” implies all addressable data, but this is a loose interpretation.

Using Modbus nomenclature, four available groups of data flags and registers contain all accessible data in a Modbus server. this includes the Modbus-master access.

- Coil status or, simply, coils are single-bit flags that represent the status of digital (Boolean) outputs from the server (slave), that is, On/Off output status. A Modbus master can both read from and write to coils.
- Input status or, simply, inputs are single-bit flags that represent the status of digital (Boolean) inputs to the server (slave), that is, On/Off input status. A Modbus master can only read inputs.
- Input registers are 16-bit registers that store data collected from the field by the Modbus server (slave). The Modbus master can read (only) input registers.
- Holding registers are 16-bit registers that store general-purpose data in the Modbus server (slave). The Modbus master can both read from and write to holding registers.

Data addresses

A Modbus device is not required to contain all four groups of data. For example, a metering device may contain only holding registers. However, for each data group implemented, an address convention is used. Requests for data (made to a device) must specify a data address (and range) of interest.

Group	Address convention
Coils	00000 - 0nnnn decimal, or 0x
Inputs	10000 - 1nnnn decimal, or 1x
Input Registers	30000 - 3nnnn decimal, or 3x
Holding Registers	40000 - 4nnnn decimal, or 4x

Data addressing (at least in decimal and hex formats) is zero-based, where the first instance of a data item, for example coil 1, is addressed as item number 0. As another example, holding register 108 is addressed as 107 decimal or 006B hex. However, it is common for a vendor to list a device’s data items using a 5-digit Modbus address, for example, holding registers starting with 40001.

Table 2. Example device register address documentation (portion)

Modbus Addr.	Units	Description	Data Type
40001	kWH	Energy Consumption, LSW	Integer (multiplication required)
40002	kWH	Energy Consumption, MSW	Integer (multiplication required)
40003	kW	Demand (power)	Integer (multiplication required)
40004	VAR	Reactive Power	Integer (multiplication required)
40005	VA	Apparent Power	Integer (multiplication required)
40006	—		Integer (multiplication required)
40007	Volts	Voltage, line to line	Integer (multiplication required)
40008	Volts	Voltage, line to neutral	Integer (multiplication required)
40009	Amps	Current	Integer (multiplication required)
40010	kW	Demand (power), phase A	Integer (multiplication required)
40011	kW	Demand (power), phase B	Integer (multiplication required)
40012		Demand (power), phase B	Integer (multiplication required)
40013	—	Power Factor, phase A	Integer (multiplication required)
40014	—	Power Factor, phase B	Integer (multiplication required)
40015	—	Power Factor, phase C	Integer (multiplication required)
—	—	—	—

Modbus Addr.	Units	Description	Data Type
40259	kWH	Energy Consumption	Float, upper 16 bits
40260	kWH	Energy Consumption	Float, lower 16 bits
40261	kW	Demand (power)	Float, upper 16 bits
40262	kW	Demand (power)	Float, lower 16 bits

Consecutive address numbering

Within any particular data group (coils, inputs, input registers, holding registers), it is typical for a Modbus device to use consecutive addresses, particularly for related data. For example, in the example, holding registers 40001-40015 are used consecutively for integer data, where each is a separate, integer, data point.

Register 40259 begins a consecutive series of holding registers used to access floating point data values. However, an address gap exists in this case. The address gap (while not necessary), was probably implemented by the device vendor for clarity.

Also, floating-point data values (being 32-bit based) require the use of two consecutive registers for each data point.

Modbus messaging supports device queries for data using both a starting address and range, which is dependent on data items being consecutively addressed. This allows for message efficiency when retrieving multiple data points, as it can be handled in one message response.

The address range for data in any data group (coils, inputs, input registers, holding registers) received in a query must be implemented by the receiving device—otherwise, it will generate an exception response. For example, a read request of holding registers 40003—40015 to the device represented by in the table receives a normal response (data values), while a similar request to registers 40003-40017 results in an illegal data address response (as holding registers 40016 and 17 are not implemented).

A Modbus driver integration makes use of consecutively addressed data in two ways:

- The **New** point window provides a Number to Add property, which assigns point register numbers in consecutive order.
- Data polling in a client device may be improved by using device polls, where data values in consecutively addressed items are requested in a single message. This reduces network messaging traffic.

Configuring register ranges

This procedure establishes the register ranges for data items in any Modbus server (slave) device. The device's proxy points must fall within these register ranges, or else they will have a fault status. This procedure works for all four register range tables: **Valid Coils Range**, **Valid Status Range** (Default), **Valid Holding Registers Range** (Default), and **Valid Input Registers Range** (Default).

As needed, for any data item range you can edit property values, add additional valid range entries, or perhaps disable ranges. For example, you could disable the **Valid Status Range** entry, so that any Modbus master queries to the device to read discrete status (inputs) would yield an exception response. As another example, you could add multiple ranges for holding registers.

Step 1. Copy the valid range component you need from the modbusSlave or modbusTcpSlave palette to the slave device in the Nav tree.

Step 2. Do one of the following:

- In the Nav tree, expand the slave network, double-click the slave Modbus device of interest, and expand the valid range and **`Default`** container.
- Right-click the valid range and click **Actions** > **Add Range**

If you expanded the property sheet, you see the four range properties: **`Enabled`**, **`Critical Data`**, **`Starting Address Offset`**, and **`Size`**.

If you used the action, the **Add Range** window opens.

By default, a slave device copied from the modbusSlave or modbusTcp palette has the same values for each of the four default register range containers: Enabled, Starting Address Offset: 1, Size: 64.

Step 3. Make whatever range entry changes are needed, and click **OK**.

Step 4. To add additional register range entries use the right-click **Add Range** action on any of the four register range containers.

Step 5. To delete a range, right-click the valid range component in the Nav tree and click **Actions** > **Clear**. This action removes all existing Modbus Register Range Entry children.

Register range example

If a holding register range is set to a starting address of 250 with a size of 75, the driver assigns an effective Modbus address range of 40250 to 40325.

NOTE: If using multiple ranges under any of the register range tables, be careful not to overlap range entries. In other words, any specific register address should be in only one range entry.

Adding server file records

Rarely, you may also wish to expose string data as Modbus files in a virtual Modbus slave device. The modbusSlave and modbusTcpSlave palettes have a component especially for this application, which you can also copy under the Modbus device.

Step 1. Open the modbusSlave or modbusTcpSlave palette in the Palette side bar.

Step 2. In the Nav side bar, expand the slave Modbus network to show the Modbus device of interest.

Step 3. From the palette, drag the Modbus File Records folder onto the Modbus device in the Nav side bar (or, into the property sheet view of that device, if open).

Step 4. In the popup **Name** window, accept the default **`Modbus File Records`** name or enter an alternate name, and click **OK**.

The folder is added under the device. By default it contains a single ModbusServerStringRecord component. You can duplicate it if multiple file record objects are needed.

Step 5. Double-click the added component to open its property sheet, and enter appropriate configuration values needed.

NOTE: You do not have to copy the entire Modbus File Records folder from the palette—this is just the easiest way to add the needed component with a descriptive parent folder. You can locate ModbusServerStringRecord components anywhere under the Modbus device. However, be aware if you copy these components under the Points container, they are not visible in any **`Modbus Server Point Manager`** view.

Chapter 6. Troubleshooting

When creating points, I get the message: `Read fault: illegal data address`**.**
Check the address in the point against the documented address for the data item.

The NumericPoint or NumericWritable I'm creating for a float or long (2-register) value reports a value of zero (0) or an impossibly large value instead of the expected value, yet the point still reports a status of {ok}.
Verify that the correct byte order settings exist for float and long values in the parent device.

Debugging messages

By enabling trace logging on a Modbus network you can monitor the query and response message cycle in a station's standard output, which results form normal data polling.

Step 1. Right-click the station in Nav tree and click **Spy**.
The **Spy Viewer** opens.

Step 2. Click the `stdout` (standard output) hyperlink.
The standard output trace log opens.

The trace log breaks out the query to show fields on separate lines, and the received (response) in a single line (in hex format).

In the case of the ModbusTCPNetwork, trace-level output shows a similar query/response message cycle from data polling, but with a slightly different format. The driver sends a 6-byte leading TCP header `000000000006` in each query, and omits the checksum byte in both sent and response messages.

Exception responses

If a Modbus slave receives a query message correctly (that is, it passes error checking), but cannot perform the required operation, the slave returns an exception response. This may happen, for instance, if the request is to read a non-existent register or coil.

An exception response message is formatted differently than a normal response, as it contains an exception code (instead of requested data). The format used is as follows:

1. Its device address, confirming to the master that it is replying to the query.
2. The function code, modified from the originally-requested function code by adding `80` hex to it (this signals the master to look for a following exception code, versus the originally-requested data).
3. The exception code number. Refers to the exception code sent by the slave, which indicates why it was unable to deliver a normal response.

   The table lists standard Modbus exception codes (01-08) plus extended codes (09-13). The driver's Modbus proxy points that reflect an exception response assume a fault status, and have a Fault Cause slot in the proxy extension that shows the name of the received exception code.

Table 3. Modbus exception codes, standard and extended

Code	Name	Meaning
01	ILLEGAL FUNCTION	The function code received in the query is not an allowable action for the slave. For example, if a FORCE SINGLE COILS (05) is received by a slave without coils, this exception code would be issued.
02	ILLEGAL DATA ADDRESS	The data address received in the query is not an allowable address for the slave. For example, if a READ INPUT REGISTERS (04) with an input register address higher than contained in the slave is received, this exception code would be issued.

Code	Name	Meaning
03	ILLEGAL DATA VALUE	A value contained in the query data field is not an allowable value for the slave. For example, if a PRESET SINGLE REGISTER (06) is received with an implied length that is incorrect, this exception code might be issued.
04	SLAVE DEVICE FAILURE	An unrecoverable error occurred while the slave was attempting to perform the requested action. For example, a READ HOLDING REGISTERS (03) is received on data that is deemed corrupted in the slave. The slave is still able to reply, however.
05	ACKNOWLEDGE	The slave has accepted the request and is processing it, but a delay is necessary before response is ready. Further polling of the slave may result in a rejected message response (06, next exception code).
06	SLAVE DEVICE BUSY	The slave is busy processing a long-duration query, or is otherwise occupied. This acknowledges to the master that the query has been received, but that the slave is too busy to respond to it.
07	NEGATIVE ACKNOWLEDGE	The slave cannot perform the requested function. An example might occur when attempting to write data in a holding register that is currently write disabled.
08	MEMORY PARITY ERROR	The slave attempted to read extended memory, but detected a parity error. A retry from the master may be successful, but the slave likely needs service.
09	noResponse	The slave is not responding to a particular query.
10 (0A)	crcError	An error-checking CRC error has been detected.
11 (0B)	otherError	The query has resulted in an uncategorized error.
12 (0C)	okNotActive	No error/No operation. The normal status of a proxy point that is not configured to poll, or of a proxy point not yet written.
13 (0D)	unknown	The slave has responded, but nothing else is known.

4. An error-check field to confirm integrity of the message, as received from the slave. If the master detects an error in the response, the master ignores the message.

Chapter 7. Components

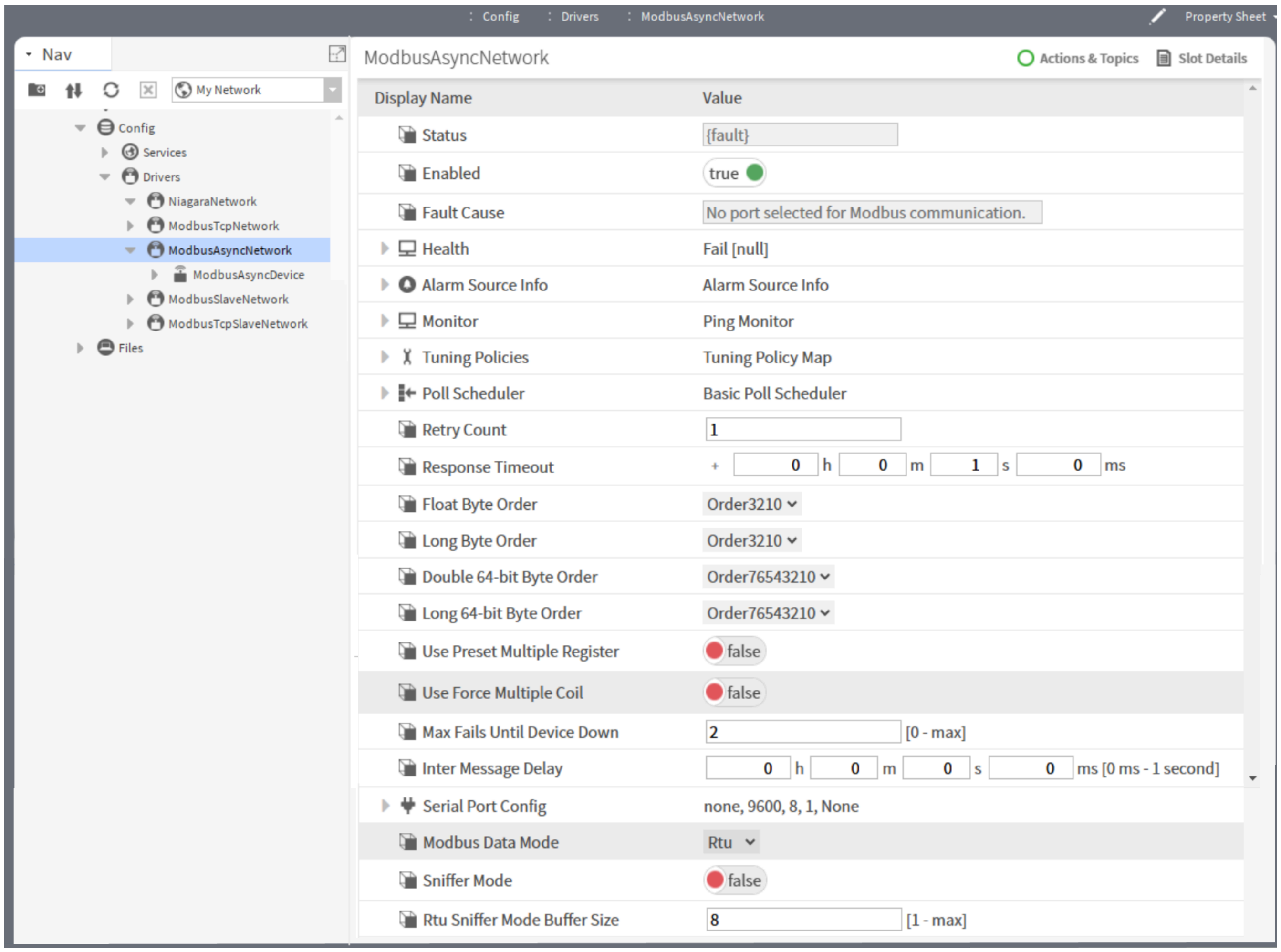
Components include services, folders and other model building blocks associated with a module. You may drag them to a **Property** or **Wire Sheet** from a palette.
Descriptions included in the following topics appear as context-sensitive help topics when accessed by:

- Right-clicking on the object and selecting **Views** > **Guide Help**
- Clicking **Help** > **Guide On Target**

ModbusAsyncNetwork

This component is the base container for one or more **ModbusAsyncDevice** components. This network component specifies the Modbus data mode (RTU or ASCII) used by the network, and houses other standard network components, including a `Serial Port Config` container (SerialHelper) to specify the serial settings used by a remote host for communications.

Figure 6. ModbusAsyncNetwork properties



Display Name	Value
Status	{fault}
Enabled	true
Fault Cause	No port selected for Modbus communication.
Health	Fail [null]
Alarm Source Info	Alarm Source Info
Monitor	Ping Monitor
Tuning Policies	Tuning Policy Map
Poll Scheduler	Basic Poll Scheduler
Retry Count	1
Response Timeout	+ 0 h 0 m 1 s 0 ms
Float Byte Order	Order3210
Long Byte Order	Order3210
Double 64-bit Byte Order	Order76543210
Long 64-bit Byte Order	Order76543210
Use Preset Multiple Register	false
Use Force Multiple Coil	false
Max Fails Until Device Down	2 [0 - max]
Inter Message Delay	0 h 0 m 0 s 0 ms [0 ms - 1 second]
Serial Port Config	none, 9600, 8, 1, None
Modbus Data Mode	Rtu
Sniffer Mode	false
Rtu Sniffer Mode Buffer Size	8 [1 - max]

You access this component by clicking **Drivers** > **ModbusAsyncNetwork** in the Nav tree. The default view is the **Modbus Async Device Manager**.

Property	Value	Description
Status	read-only	Reports the condition of the entity or process at last polling. `{ok}` indicates that the component is licensed and polling successfully. `{down}` indicates that the last check was unsuccessful, perhaps because of an incorrect property, or possibly loss of network connection. `{disabled}` indicates that the **`Enable`** property is set to `false`. `{fault}` indicates another problem. Refer to **`Fault Cause`** for more information.
Enabled	`true` (default) or `false`	Activates (`true`) and deactivates (`false`) use of the object (network, device, point, component, table, schedule, descriptor, etc.).
Fault Cause	read-only	Indicates the reason why a system object (network, device, component, extension, etc.) is not working (in fault). This property is empty unless a fault exists.
Health	read-only	Reports the status of the network, device or component. This advisory information, including a time stamp, can help you recognize and troubleshoot problems but it provides no direct management controls. The *Niagara Drivers Guide* documents the these properties.
Alarm Source Info	additional properties	Contains a set of properties for configuring and routing alarms when this component is the alarm source. For property descriptions, refer to the *Niagara Alarms Guide*

Property	Value	Description
Monitor	Ping Monitor	Configures a network's ping mechanism, which verifies network health. This includes verifying the health of all connected objects (typically, devices) by pinging each device at a repeated interval. The *Niagara Drivers Guide* documents these properties.
Tuning Policies	additional properties	Configures network rules for evaluating both write requests to writable proxy points as well as the acceptable freshness of read requests. For more information, refer to the *Niagara Drivers Guide*.
Poll Scheduler	additional properties	Configures the frequency with which the driver polls points and devices. “Poll Service properties” in the *Niagara Drivers Guide* documents these properties.
Retry Count	number	Configures how many times to repeat a network read request if no response is received before the response timeout interval elapses.
Response Timeout	hours, minutes, seconds	Defines the length of time to wait before a communication times out.
Float Byte Order	drop-down list	Specifies the byte-order used to assemble or receive floating-point (32-bit) values in messages. Choices reflect two alternate methods, where numerals 0, 1, 2, and 3 represent the least significant byte to most significant byte: • `Order3210` – Most significant byte first, or big-endian, it is the default.

Property	Value	Description
		• `Order1032` – Bytes transmitted in a 1,0,3,2 order, or little–endian. • `Order0123` – Bytes transmitted in a 0,1,2,3 order, or little-endian.
Long Byte Order	drop-down list	Specifies the byte-order used to assemble or receive long integer (32-bit) values in messages. Choices reflect two alternate methods, where numerals 0, 1, 2, and 3 represent the least significant byte to most significant byte: • `Order3210` - Most significant byte first, or big-endian, it is the default. • `Order1032` - Bytes transmitted in a 1,0,3,2 order, or little-endian. • `Order0123` - Bytes transmitted in a 0,1,2,3 order, or little-endian. **NOTE:** Float or long values received in incorrect byte order may appear abnormally big, or not at all.
Double 64-bit Byte Order	drop-down list	Specifies the byte-order used to assemble or receive receives double-precision floating point (64-bit) values in messages. Choices reflect in 8 corresponding byte order options, where numerals 0, 1, 2, and 3 represent the least significant byte to most significant byte. When selecting the byte order options, it is required that the Float Byte Order and Long Byte Order are both set to `Order0123` to implement this configuration effectively: • `Order76543210` – Most significant byte first, or big-endian order where the most significant byte is transmitted first (from 7

Property	Value	Description
		down to 0). • `Order 67452301` – Bytes transmitted in a order 6,7,4,5, etc or big-endian format involves swapping the bytes. • `Order54761032` – Bytes transmitted in a order 5, 4, 7, 6, etc or big-endian format but with each pair of bytes swapped (e.g., switching the first two bytes and the next two bytes). • `Order45670123` – Bytes are transmitted in order 4, 5, 6, 7, etc or big-endian with both words and bytes swapped. • `Order01234567` – Bytes are transmitted in order 0, 1, 2, 3, etc or little-endian (LE) arrangement where the bytes are transmitted in direct order from least significant to most significant (from 0 to 7). • `Order10325476` – Bytes are transmitted in order 1, 0, 3, 2, etc or little-endian format where the bytes are swapped during transmission. • `Order23016745` – Bytes are transmitted in order 2, 3, 0, 1, etc or little-endian format but with each pair of bytes swapped (e.g., switching the first two bytes and the next two bytes). • `Order32107654` – Bytes are transmitted in order 3, 2, 1, 0, etc or little-endian format, with both words and bytes swapped.
Long 64-bit Byte Order	drop-down list	Specifies the byte-order used to assemble or receive long integer (64-bit) values in messages. Choices reflect in 8 corresponding byte order options, where numerals 0, 1, 2,

Property	Value	Description
		and 3 represent the least significant byte to most significant byte. When selecting the byte order options, it is required that the Float Byte Order and Long Byte Order are both set to `Order0123` to implement this configuration effectively: • `Order76543210` – Most significant byte first, or big-endian (BE) order where the most significant byte is transmitted first (from 7 down to 0). • `Order 67452301` – Bytes transmitted in a order 6,7,4,5, etc or big-endian format involves swapping the bytes. • `Order54761032` – Bytes transmitted in a order 5, 4, 7, 6, etc or big-endian format but with each pair of bytes swapped (e.g., switching the first two bytes and the next two bytes). • `Order45670123` – Bytes are transmitted in order 4, 5, 6, 7, etc. or big-endian with both words and bytes swapped. • `Order01234567` – Bytes are transmitted in order 0, 1, 2, 3, etc or little-endian (LE) arrangement where the bytes are transmitted in direct order from least significant to most significant (from 0 to 7). • `Order10325476` – Bytes are transmitted in order 1, 0, 3, 2, etc or little-endian format where the bytes are swapped during transmission. • `Order23016745` – Bytes are transmitted in order 2, 3, 0, 1, etc or little-endian format but with each pair of bytes swapped (e.g., switching the first two bytes

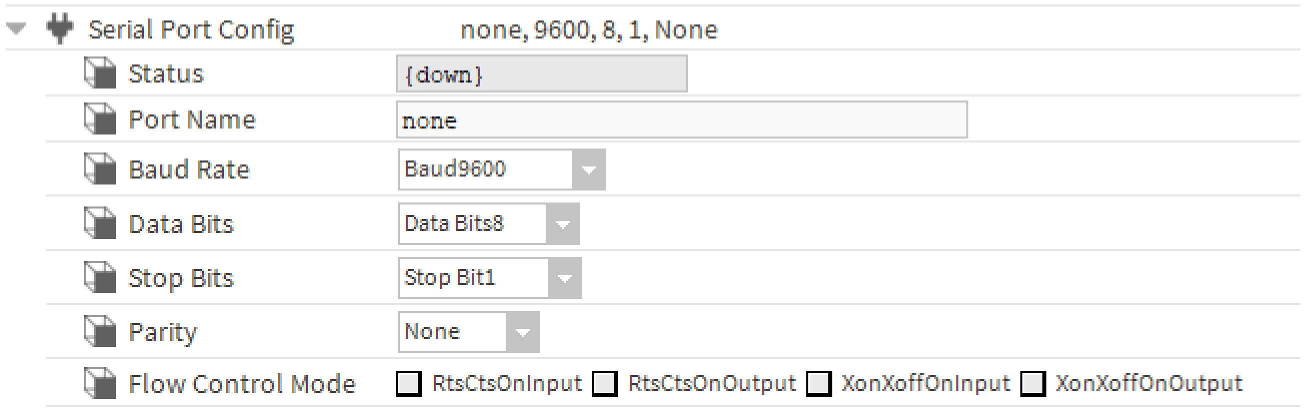
Property	Value	Description
		and the next two bytes). • `Order32107654` – Bytes are transmitted in order 3, 2, 1, 0, etc or little-endian format, with both words and bytes swapped.
Use Preset Multiple Register	`true` or `false` (default)	Specifies whether to use function code 16 (**`Preset Multiple Registers`**) instead of function code 06 (**`Preset Single Register`**) when writing to registers. The default is `false`, where function code 06 (**`Preset Single Register`**) is used. This property depends on the Modbus function codes supported by child devices, which (if available) provide alternative options in Modbus messaging. Function code 16 support (**`Preset Multiple Registers`**) is available in devices (`true` or `false`). The default is `false`, where function code **`Preset Single Register`** is in place.
Use Force Multiple Coil	`true` or `false` (default)	Specifies whether to use function code 15 (Force Multiple Coils) instead of function code 05 (Force Single Coil) when writing to coils. The default is `false`, where function code 05 (Force Single Coil) is used. This property depends on the Modbus function codes supported by child devices, which (if available) provide alternative options in Modbus messaging. Function code 15 support (Preset Multiple Coils) is available in devices (true or false). The default is `false`, where function code **`Preset Single Coil`** is in place.
Max Fails Until Device Down	number (defaults to 2)	Defines how many times a communication may fail before the device with which the system

Property	Value	Description
		is communicating is considered to be down.
InterMessageDelay	hours, minutes and seconds (defaults to 1 second)	Configures the time delay between messages.
Serial Port Config	additional properties	Refer to Serial Port Config
Modbus Data Mode	drop-down list (defaults to `Rtu`)	Selects the type of serial network. `Rtu` (Remote Terminal Unit) `Ascii` (American Standard Code for Information Interchange)
Sniffer Mode	`true` or `false` (default)	Usually left at the default unless there is a particular reason to change it.
Rtu Sniffer Mode Buffer Size	number (defaults to `8`)	Usually left at the default unless there is a particular reason to change it.

Serial Port Config

Specifies the serial port/communications setup required to communicate to the serial Modbus devices.

Figure 7. Serial Port Config properties



Serial Port Config	none, 9600, 8, 1, None
Status	{down}
Port Name	none
Baud Rate	Baud9600
Data Bits	Data Bits8
Stop Bits	Stop Bit1
Parity	None
Flow Control Mode	☐ RtsCtsOnInput ☐ RtsCtsOnOutput ☐ XonXoffOnInput ☐ XonXoffOnOutput

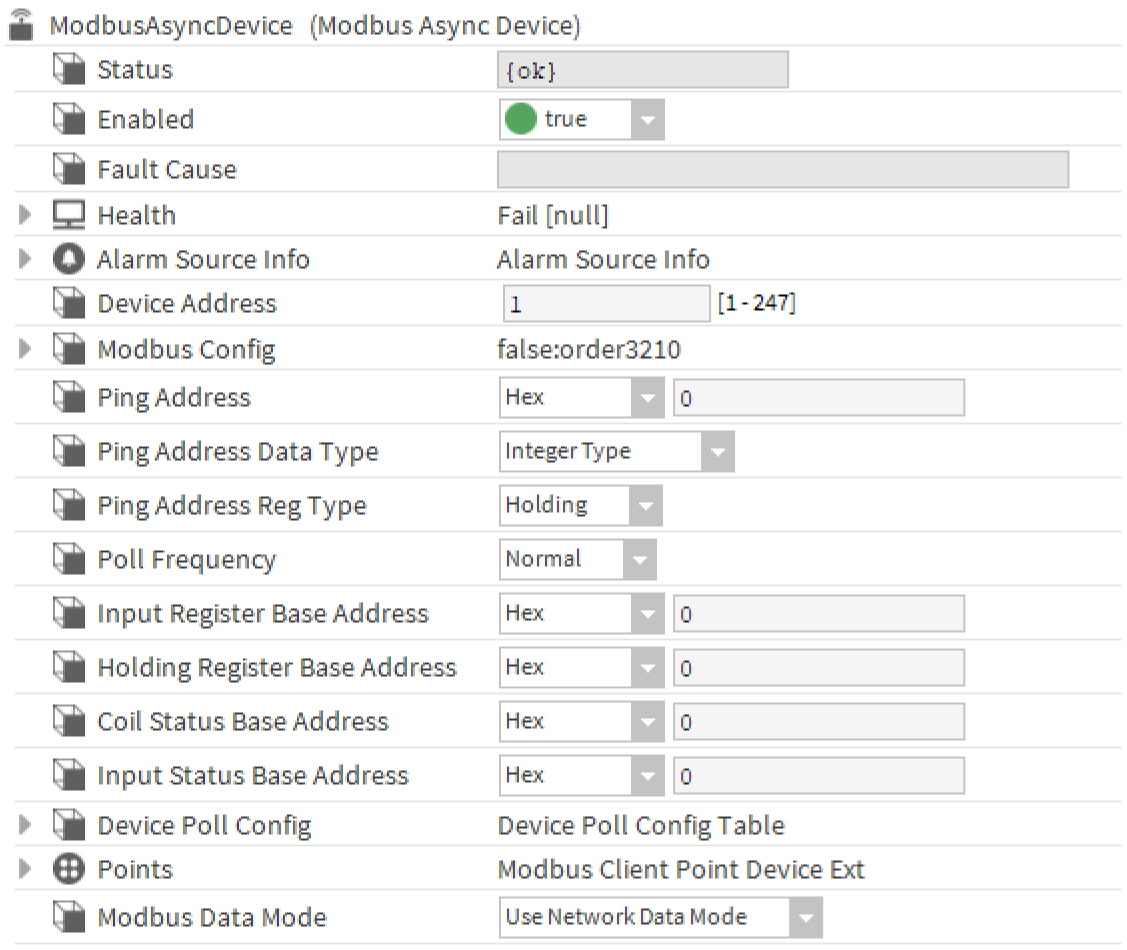
Property	Value	Description
Status	read-only	Reports the condition of the entity or process at last polling. `{ok}` indicates that the

Property	Value	Description
		component is licensed and polling successfully. `{down}` indicates that the last check was unsuccessful, perhaps because of an incorrect property, or possibly loss of network connection. `{disabled}` indicates that the **`Enable`** property is set to `false`. `{fault}` indicates another problem. Refer to **`Fault Cause`** for more information.
Port Name	text (defaults to `none`)	Defines the communication port to use: `none`, `COM2` or `COM3`.
Baud Rate	drop-down list (defaults to `Baud9600`)	Defines communication speed in bits per second.
Data Bits	drop-down list (defaults to `Data Bits8`)	Defines the number of bits required to encode a character (a byte).
Stop Bits	drop-down list (defaults to `Stop Bit1`)	Defines the number of bits that indicate the end of a character.
Parity	drop-down list (defaults to `None`)	Defines the odd or even requirement of a transmitted byte of data for the purpose of error detection.
Flow Control Mode	check box	Using the selected Modbus protocol, manages the efficient transmission of data between two devices.

ModbusAsyncDevice

This component represents a Modbus serial (async) device under a ModbusAsyncNetwork, and provides client access by the station (acting as Modbus master). In addition to the typical device components, it contains properties to specify the device's Modbus address, data mode (RTU or ASCII), and other properties, including slots to specify the base address for Modbus data items (holding registers, input registers, inputs, coils), plus a DevicePollConfigTable for device polling.

Figure 8. ModbusAsyncDevice properties



ModbusAsyncDevice (Modbus Async Device)	
Status	{ok}
Enabled	true
Fault Cause	
Health	Fail [null]
Alarm Source Info	Alarm Source Info
Device Address	1 [1 - 247]
Modbus Config	false:order3210
Ping Address	Hex 0
Ping Address Data Type	Integer Type
Ping Address Reg Type	Holding
Poll Frequency	Normal
Input Register Base Address	Hex 0
Holding Register Base Address	Hex 0
Coil Status Base Address	Hex 0
Input Status Base Address	Hex 0
Device Poll Config	Device Poll Config Table
Points	Modbus Client Point Device Ext
Modbus Data Mode	Use Network Data Mode

You access these properties by expanding **Drivers** > **ModbusAsyncNetwork** and double-clicking the **ModbusAsyncDevice** in the Nav tree.

In addition to the standard properties (Status, Enabled, Fault Cause, Health and Alarm Source Info), these properties support the Modbus async device:

Property	Value	Description
Device Address	number from 1 to 247	Defines the unique number that identifies the current device object on the network.
Modbus Config	additional properties	Refer to Modbus Config .
Ping Address, Address Format	Hex (default), Decimal, or Modbus	Selects the format of an address used to automatically configure unique register addresses for the device's points. The driver merges this address with the point address.

Property	Value	Description
		For an example, refer to Base addresses **[/normalized/reference/reference/reference/refbody/properties/property/propdesc/title {"- topic/title "})** Unique point address example **(title]**.
Ping Address, Address	number	Defines the base address to use.
Ping Address Data Type	drop-down list	Defines the type of numeric data.
Ping Address Reg Type	drop-down list	Defines the type of register.
Poll Frequency	drop-down list	Configures how frequently the system polls proxy points.
Input Register Base Address, Address Format	Hex (default), Decimal, or Modbus	Selects the format of an address used to automatically configure unique register addresses for the device's points. The driver merges this address with the point address. For an example, refer to Base addresses **[/normalized/reference/reference/reference/refbody/properties/property/propdesc/title {"- topic/title "})** Unique point address example **(title]**
Input Register Base Address, Address	number	Defines the base address to use.
Holding Register Base Address, Address Format	Hex (default), Decimal, or Modbus	Selects the format of an address used to automatically configure unique register addresses for the device's points. The driver merges this address with the point address. For an example, refer to Base addresses **[/normalized/reference/reference/reference/refbody/properties/property/propdesc/title {"- topic/title "})** Unique point address example **(title]**.
Holding Register Base Address, Address	number	Defines the base address to use.
Coil Status Base Address, Address Format	Hex (default), Decimal, or Modbus	Selects the format of an address used to automatically configure unique register addresses for the device's points. The driver merges this address with the point address.

Property	Value	Description
		For an example, refer to Base addresses **[/normalized/reference/reference/reference/refbody/properties/property/propdesc/title {"- topic/title "})** Unique point address example **(title]**.
Coil Status Base Address, Address	text	Defines the base address to use.
Input Status Base Address Format	Hex (default), Decimal, or Modbus	Selects the format of an address used to automatically configure unique register addresses for the device's points. The driver merges this address with the point address. For an example, refer to Base addresses **[/normalized/reference/reference/reference/refbody/properties/property/propdesc/title {"- topic/title "})** Unique point address example **(title]**.
Input Status Base Address	number	Defines the base address to use.
Device Poll Config	additional properties	Device Poll Config
Points	additional properties	Points (client device)

Base addresses

Defining a base address provides a quick way to configure unique point addresses. For example, assume you have 10 of the same devices each with 20 points. Configuring a unique address for 200 points could take a substantial amount of time. To speed configuration, you could configure a single device with all its point addresses set to 01–20, duplicate that device 10 times, then change the base address(es) for each device. The result: all points have unique addresses.

Table 4. Unique point address example

Device	Base Address	Resulting point addresses
1	100	101, 102, 103, ... 120
2	200	201, 202, 203, ... 220
3	300	302, 302, 303, ... 320
4	400	401, 402, 493, ... 420
etc.	etc.	etc.

This works the same for all the base address properties: `Input Register`, `Holding Register`, `Coil Status`, and `Input Status`.

Modbus Config

Each client Modbus device object (ModbusAsyncDevice, ModbusTCPDevice, and ModbusTCPGatewayDevice) has an associated **Modbus Config** container slot to override these network-wide defaults. These properties adjust the settings for message transactions to (and from) only that device.

Property	Value	Description
Override Network	`true` or `false` (default)	Determines which values to use for these properties: **`Float Byte Order`**, **`Long Byte Order`** and **`Use Force Multiple Coil`**. `false` selects the network-level values as configured by the ModbusAsyncNetwork component. `true` selects the values defined by the **Modbus Config** container slot.
Float Byte Order	drop-down list	Specifies the byte-order used to assemble or receive floating-point (32-bit) values in messages. Choices reflect two alternate methods, where numerals 0, 1, 2, and 3 represent the least significant byte to most significant byte: • `Order3210` – Most significant byte first, or big-endian, it is the default. • `Order1032` – Bytes transmitted in a 1,0,3,2 order, or little–endian. • `Order0123` – Bytes transmitted in a 0,1,2,3 order, or little-endian.
Long Byte Order	drop-down list	Specifies the byte-order used to assemble or receive long integer (32-bit) values in messages. Choices reflect two alternate methods, where numerals 0, 1, 2, and 3 represent the least significant byte to most significant byte: • `Order3210` - Most significant byte first, or big-endian, it is the default. • `Order1032` - Bytes transmitted in a 1,0,3,2 order, or little-endian. • `Order0123` - Bytes transmitted in a 0,1,2,3 order, or little-endian.

Property	Value	Description
		NOTE: Float or long values received in incorrect byte order may appear abnormally big, or not at all.
Double 64-bit Byte Order	drop-down list	Specifies the byte-order used to assemble or receive receives double-precision floating point (64-bit) values in messages. Choices reflect in 8 corresponding byte order options, where numerals 0, 1, 2, and 3 represent the least significant byte to most significant byte. When selecting the byte order options, it is required that the Float Byte Order and Long Byte Order are both set to `Order0123` to implement this configuration effectively: • `Order76543210` – Most significant byte first, or big-endian order where the most significant byte is transmitted first (from 7 down to 0). • `Order 67452301` – Bytes transmitted in a order 6,7,4,5, etc or big-endian format involves swapping the bytes. • `Order54761032` – Bytes transmitted in a order 5, 4, 7, 6, etc or big-endian format but with each pair of bytes swapped (e.g., switching the first two bytes and the next two bytes). • `Order45670123` – Bytes are transmitted in order 4, 5, 6, 7, etc or big-endian with both words and bytes swapped. • `Order01234567` – Bytes are transmitted in order 0, 1, 2, 3, etc or little-endian (LE) arrangement where the bytes are transmitted in direct order from least significant to most

Property	Value	Description
		significant (from 0 to 7). • `Order10325476` – Bytes are transmitted in order 1, 0, 3, 2, etc or little-endian format where the bytes are swapped during transmission. • `Order23016745` – Bytes are transmitted in order 2, 3, 0, 1, etc or little-endian format but with each pair of bytes swapped (e.g., switching the first two bytes and the next two bytes). • `Order32107654` – Bytes are transmitted in order 3, 2, 1, 0, etc or little-endian format, with both words and bytes swapped.
Long 64-bit Byte Order	drop-down list	Specifies the byte-order used to assemble or receive long integer (64-bit) values in messages. Choices reflect in 8 corresponding byte order options, where numerals 0, 1, 2, and 3 represent the least significant byte to most significant byte. When selecting the byte order options, it is required that the Float Byte Order and Long Byte Order are both set to `Order0123` to implement this configuration effectively: • `Order76543210` – Most significant byte first, or big-endian (BE) order where the most significant byte is transmitted first (from 7 down to 0). • `Order 67452301` – Bytes transmitted in a order 6,7,4,5, etc or big-endian format involves swapping the bytes. • `Order54761032` – Bytes transmitted in a order 5, 4, 7, 6, etc or big-endian format but with each pair of bytes swapped (e.g.,

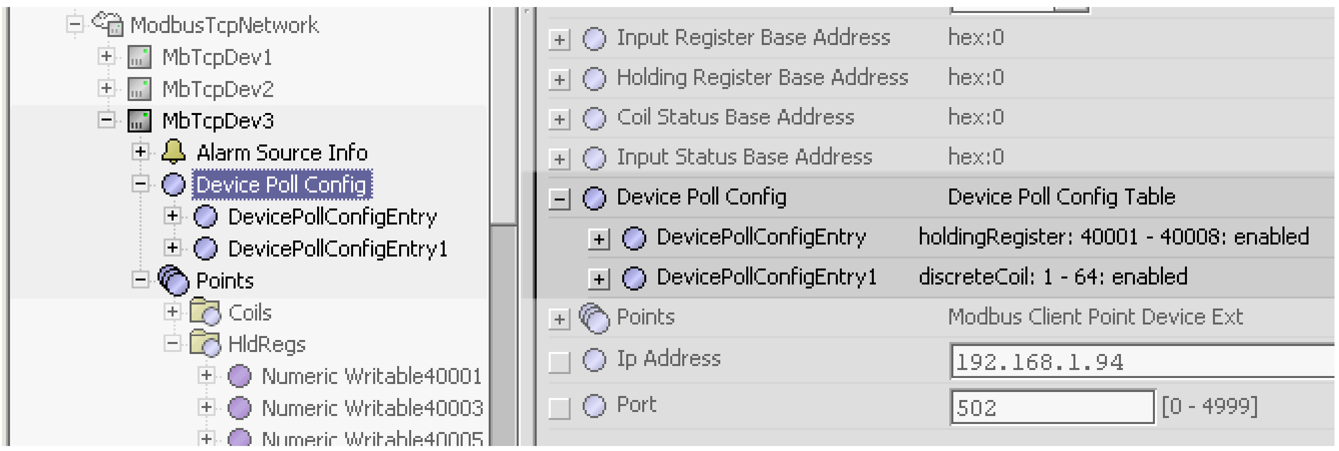
Property	Value	Description
		switching the first two bytes and the next two bytes). • `Order45670123` – Bytes are transmitted in order 4, 5, 6, 7, etc. or big-endian with both words and bytes swapped. • `Order01234567` – Bytes are transmitted in order 0, 1, 2, 3, etc or little-endian (LE) arrangement where the bytes are transmitted in direct order from least significant to most significant (from 0 to 7). • `Order10325476` – Bytes are transmitted in order 1, 0, 3, 2, etc or little-endian format where the bytes are swapped during transmission. • `Order23016745` – Bytes are transmitted in order 2, 3, 0, 1, etc or little-endian format but with each pair of bytes swapped (e.g., switching the first two bytes and the next two bytes). • `Order32107654` – Bytes are transmitted in order 3, 2, 1, 0, etc or little-endian format, with both words and bytes swapped.
Use Force Multiple Coil	`true` or `false` (default)	Specifies whether to use function code 15 (Force Multiple Coils) instead of function code 05 (Force Single Coil) when writing to coils. The default is `false`, where function code 05 (Force Single Coil) is used. This property depends on the Modbus function codes supported by child devices, which (if available) provide alternative options in Modbus messaging. Function code 15 support (Preset Multiple Coils) is available in devices (true or false). The default is `false`, where function

Property	Value	Description
		code **`Preset Single Coil`** is in place.
Use Preset Multiple Register	`true` or `false` (default)	Specifies whether to use function code 16 (**`Preset Multiple Registers`**) instead of function code 06 (**`Preset Single Register`**) when writing to registers. The default is `false`, where function code 06 (**`Preset Single Register`**) is used. This property depends on the Modbus function codes supported by child devices, which (if available) provide alternative options in Modbus messaging. Function code 16 support (**`Preset Multiple Registers`**) is available in devices (`true` or `false`). The default is `false`, where function code **`Preset Single Register`** is in place.

Device Poll Config

This frozen slot under a client **ModbusAsyncDevice**, **ModbusTcpGatewayDevice** contains a table of device poll configuration entries, which specify the device-level polling of points.

Figure 9. Device Poll Config Properties



Input Register Base Address	hex:0
Holding Register Base Address	hex:0
Coil Status Base Address	hex:0
Input Status Base Address	hex:0
Device Poll Config	Device Poll Config Table
DevicePollConfigEntry	holdingRegister: 40001 - 40008: enabled
DevicePollConfigEntry1	discreteCoil: 1 - 64: enabled
Points	Modbus Client Point Device Ext
Ip Address	192.168.1.94
Port	502 [0 - 4999]

You access these properties by double-clicking the device node in the Nav tree and expanding **`Device Poll Config`**.

By default (initially) the **`Device Poll Config`** container is empty, but it can hold one or more entry children, which configure and enable device polling.

Property	Value	Description
DevicePollConfigEntry	additional properties	DevicePollConfigEntry

Actions

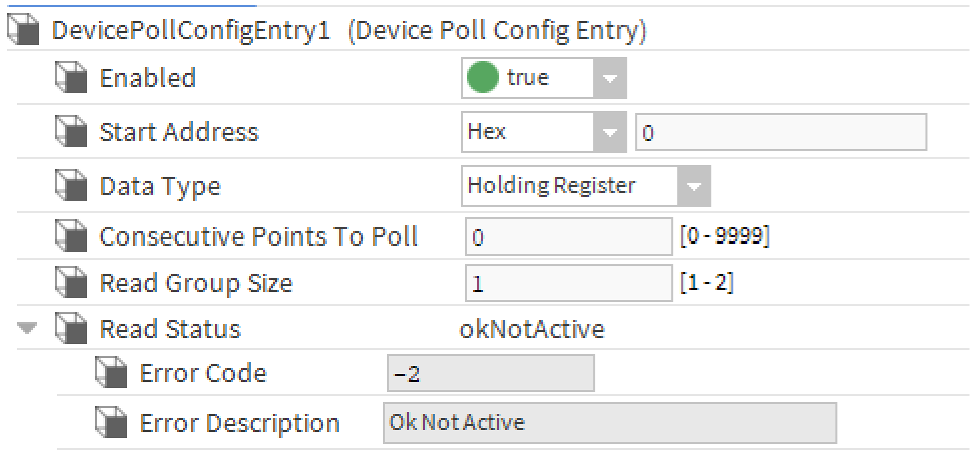
The Device Poll Config table has two available actions:

- `Learn Optimum Device Poll Config` automatically creates child DevicePollConfigEntry components based upon the current collection of Modbus proxy points.
- `Clear` removes all existing child DevicePollConfigEntry components.

DevicePollConfigEntry

This component configures device-level polling for consecutive proxy points, where one or more of these components may be under the **DevicePollConfigTable** (**`Device Poll Config`**) slot of a client **ModbusDevice**. Properties specify the starting Modbus address and number of points to poll.

Figure 10. Device Poll Config Entry properties



DevicePollConfigEntry1 (Device Poll Config Entry)

Enabled	● true	
Start Address	Hex	0
Data Type	Holding Register	
Consecutive Points To Poll	0	[0 - 9999]
Read Group Size	1	[1 - 2]
Read Status	okNotActive	
Error Code	-2	
Error Description	Ok Not Active	

You access these properties by expanding **ModbusAsyncDevice** > **DevicePollConfig** > **DevicePollConfigEntry** in the Nav tree.

Property	Value	Description
Enabled	`true` (default) or `false`	If set to `false`, associated proxy points use individual point polls instead.
Start Address	additional properties	Refer to Start Address .
Data Type	drop-down list	Selects the location of the data in

Property	Value	Description
		the slave device. Coils store on/off values. Registers store numeric values. Each coil or contact is 1 bit with an assigned address between 0000 and 270E. Each register is one word (16 bits, 2 bytes) as well as a data address between 0000 and 270E.
Consecutive Points to Poll	number from 0 to 9999	Indicates how many points to poll beginning with the `Start Address`.
Read Group Size	number (1 or 2) (defaults to 1)	Usually 1 unless all data items are 2-register types (float or long values), although a 1 works the same, provides consecutive points to poll is really consecutive registers.
Read Status	additional properties	Reports a numerical `Error Code` (0-2), and a corresponding `Error Description`.

Start Address

First data item address, including format and numerical address.

Property	Value	Description
Address Format	Hex (default), Decimal, or Modbus	Selects the format of an address used to automatically configure unique register addresses for the device's points. The driver merges this address with the point address.
Address	text	Defines the base address to use.

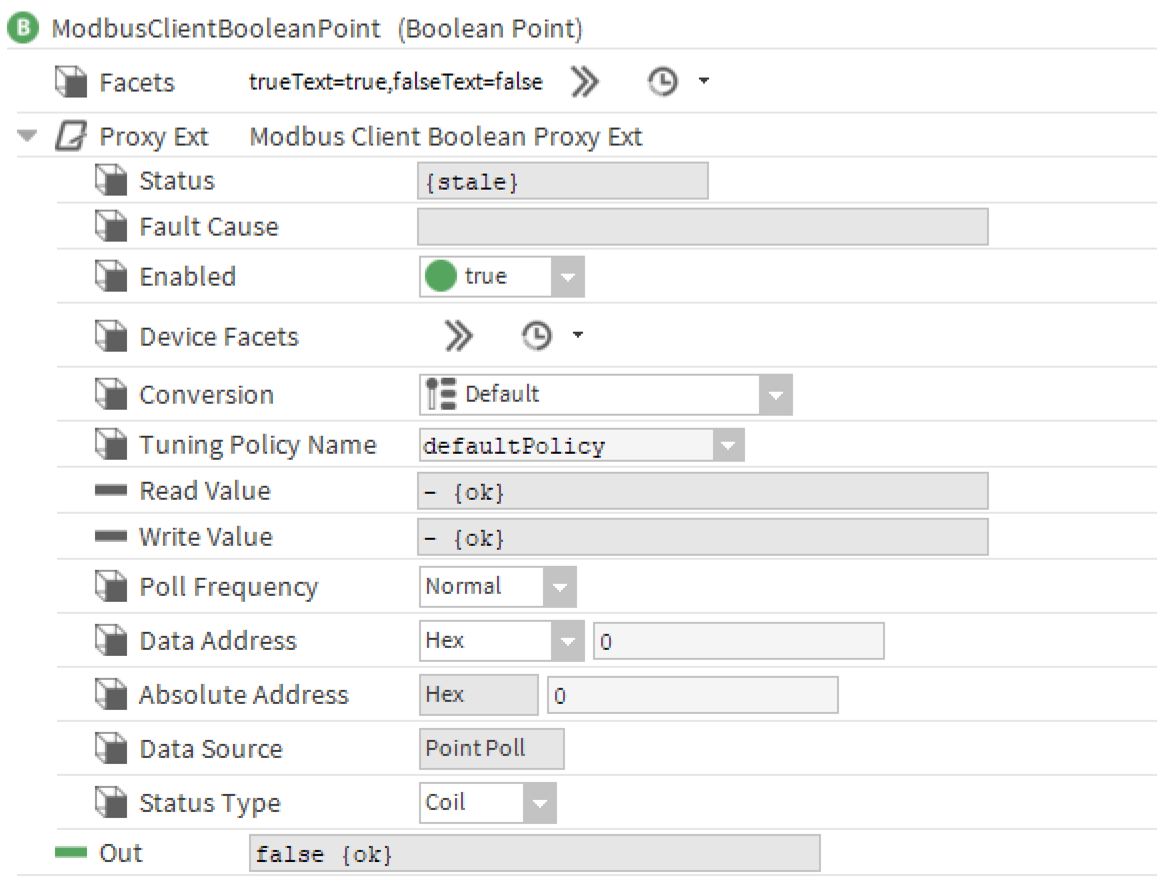
Points (client device)

This **Points** component extension is the Modbus client implementation of PointDeviceExt, and is a frozen extension under every **ModbusAsyncDevice** and **ModbusTcpDevice**. Its primary view is the **ModbusClientPointManager**.

Proxy Ext (Client Boolean)

This is the proxy extension for either a ModbusClientBooleanPoint (BooleanPoint) or ModbusClientBooleanWritable (BooleanWritable). It contains necessary information to poll (read) a status data value from a client Modbus device.

Figure 11. Client Boolean Proxy Ext properties



ModbusClientBooleanPoint (Boolean Point)	
Facets	trueText=true,falseText=false
Proxy Ext	Modbus Client Boolean Proxy Ext
Status	{stale}
Fault Cause	
Enabled	true
Device Facets	
Conversion	Default
Tuning Policy Name	defaultPolicy
Read Value	- {ok}
Write Value	- {ok}
Poll Frequency	Normal
Data Address	Hex 0
Absolute Address	Hex 0
Data Source	Point Poll
Status Type	Coil
Out	false {ok}

You access these properties by expanding **ModbusAsynNetwork** > **ModbusAsyncDevice** > **Points** in the Nav tree and double-clicking the ModbusClientBooleanPoint (BooleanPoint) or ModbusClientBooleanWritable (BooleanWritable).

In addition to the standard properties (Status, Fault Cause and Enabled), these properties are unique to the Client Boolean proxy extension.

Property	Value	Description
Device Facets	read-only	Determine how values are formatted for display depending on the context and the type of data. Examples include engineering units and decimal precision for numeric types, and descriptive value (state) text for boolean and enum types. With the exception of proxy points (with possible defined

Property	Value	Description
		device facets), point facets do not affect how the framework processes the point's value. Besides control points, various other components have facets too. For example, many kitControl and schedule components have facets. Details about point facets apply to these components too, unless especially noted. You access facets by clicking an **Edit** button or a chevron >>. Both open an **Edit Facets** window.
Conversion	drop-down list	Selects the units to use when converting values from the device facets to point facets. `Default` automatically converts similar units (such as Fahrenheit to Celsius) within the proxy point. **NOTE:** In most cases, the standard `Default` is best. `Linear` applies to voltage input, resistive input and voltage output writable points. Works with linear-acting devices. You use the Scale and Offset properties to convert the output value to a unit other than that defined by device facets. `Linear With Unit` is an extension to the existing linear conversion property. This specifies whether the unit conversion should occur on "Device Value" or "Proxy Value". The new linear with unit convertor, will have a property to indicate whether the unit conversion should take place before or after the scale/offset conversion. `Reverse Polarity` applies only to Boolean input and relay output

Property	Value	Description
		writable points. Reverses the logic of the hardware binary input or output. `500 Ohm Shunt` applies to voltage input points only. It reads a 4-to-20mA sensor, where the Ui input requires a 500 ohm resistor wired across (shunting) the input terminals. `Tabular Thermistor` applies to only a Thermistor input point and involves a custom resistance-to-temperature value response curve for Type 3 Thermistor temperature sensors. `Thermistor Type 3` applies to an Thermistor Input point, where this selection provides a “built-in” input resistance-to-temperature value response curve for Type 3 Thermistor temperature sensors. `Generic Tabular` applies to non-linear support for devices other than for thermistor temperature sensors with units in temperature. Generic Tabular uses a lookup table method similar to the “Thermistor Tabular” conversion, but without predefined output units.
Turning Policy Name	drop-down list (defaults to **`Default Policy`**)	Specifies the tuning policy to use for the proxy point.
Read Value	read-only	Reports the value read by the driver from the device and formatted based on device facets. This value agrees with point facets.
Write Value	read-only	Displays the last value written using device facets.
Poll Frequency	drop-down list	Configures how frequently the system polls proxy points.
Data Address	drop-down list (defaults to `Modbus`)	Specifies the address of the

Property	Value	Description

		polled data item (prior to any offset address change as a result of using device-level Base Address), as a combination of the address format and the numerical address expressed in the selected format. The formats are:
		`Modbus`
		`Hex`
		`Decimal`
		For example, the following are all equivalent addresses:
		• Modbus, 40012
		• Hex, 0B
		• Decimal, 11
		.
		NOTE: If you use the `Hex` or `Decimal` format for most read-only points you need to specify the **`Reg Type`** property, to clarify whether you are using a holding register or an input register.

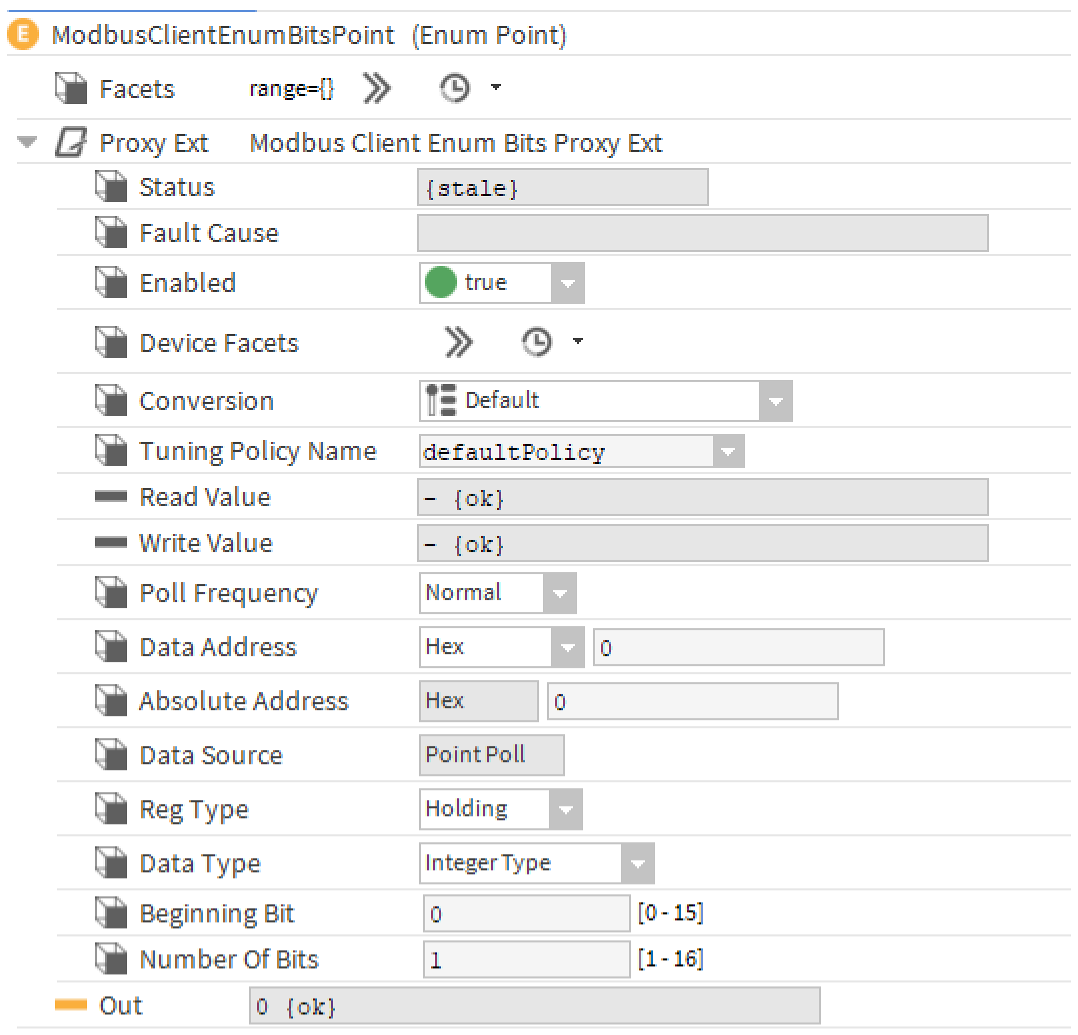
Absolute Address	read-only	Differs from **`Data Address`** only if using device Base Addresses. It is the sum of the **`Data Address`** value and the associated Base Address value (as configured in the parent Modbus device). This is the actual address that will be used when polling for this discrete data point from the actual Modbus device. The address of the polled data item is a combination of: • Address Format — either Modbus (default), Hex, or Decimal. • Address — numerical address, expressed in the selected **`Data Address`** format.
Data Source	read-only	Identifies where the data came from, such as "Point Poll."
Status Type	drop-down list	Selects between `Coil` or `Input` to define the type to read. `Coil` is the only valid option (the master

Property	Value	Description
		cannot write to Modbus inputs). Selection is only necessary if the **Data Address** format is set to `Hex` or `Decimal`. The Modbus **Address Format**, if used, automatically sets this property value.
Out	read-only	Displays the current value of the proxy point including facets and status. The value depends on the type of control point. Facets define how the value displays, including the value’s number of decimal places, engineering units, or text descriptors for Boolean/enum states. You can edit point facets to poll for additional properties, such as the native statusFlags and/or priorityArray level. Status reports the current health and validity of the value. Status is specified by a combination of status flags, such as `fault`, `overridden`, `alarm`, and so on. If no status flag is set, status is considered normal and reports `{ok}`.

Proxy Ext (Client Enum Bits)

This is the proxy extension for either a ModbusClientEnumBitsPoint (Enum Bits Point) or ModbusClientEnumBitsWritable (Enum Bits Writable). It supports reading both Modbus holding register or input register values, and extracting bits specified by the **Beginning Bit** and **Number of Bits** properties. The combination of these bits is the point’s value, as a StatusEnum. The writable variant writes the point’s (ordinal, integer) value into the raw register bits at the specified **Beginning Bit** relative position. Like other Modbus proxy points, both point-level or device-level polling is supported.

Figure 12. Modbus Client Enum Bits properties


ModbusClientEnumBitsPoint (Enum Point)

Facets	range={}
Proxy Ext	Modbus Client Enum Bits Proxy Ext
Status	{stale}
Fault Cause	
Enabled	true
Device Facets	
Conversion	Default
Tuning Policy Name	defaultPolicy
Read Value	- {ok}
Write Value	- {ok}
Poll Frequency	Normal
Data Address	Hex 0
Absolute Address	Hex 0
Data Source	Point Poll
Reg Type	Holding
Data Type	Integer Type
Beginning Bit	0 [0 - 15]
Number Of Bits	1 [1 - 16]
Out	0 {ok}

You access these properties by expanding **ModbusAsynNetwork** > **ModbusAsyncDevice** > **Points** in the Nav tree and double-clicking ModbusClientEnumBitsPoint or ModbusClientEnumBitsWritable.

In addition to the standard properties (Status, Fault Caues and Enabled), these properties are specific to the Client Enum Bits proxy extension:

Property	Value	Description
Facets — Boolean	*trueText* (default) or *falseText*	Define the text to display for the Boolean values: • `trueText` is the text to display when output is true • `falseText` is the text to display when output is false. For example, the facet `trueText`

Property	Value	Description
		could display “ON” and the facet `falseText` “OFF.” You view Facets on the Slot Sheet and edit them from a component **Property Sheet** by clicking the >> icon to display the **Config Facets** window.
Turning Policy Name	drop-down list (defaults to **`Default Policy`**)	Specifies the tuning policy to use for the proxy point.
Read Value	read-only	Reports the value read by the driver from the device and formatted based on device facets. This value agrees with point facets.
Write Value	read-only	Displays the last value written using device facets.
Poll Frequency	drop-down list	Configures how frequently the system polls proxy points.

Data Address	drop-down list (defaults to `Modbus`)	Specifies the address of the polled data item (prior to any offset address change as a result of using device-level Base Address), as a combination of the address format and the numerical address expressed in the selected format. The formats are:
		
		`Modbus`
		
		`Hex`
		
		`Decimal`
		
		For example, the following are all equivalent addresses:
		• Modbus, 40012
		• Hex, 0B
		• Decimal, 11
		.

Property	Value	Description
		NOTE: If you use the `Hex` or `Decimal` format for most read-only points you need to specify the **`Reg Type`** property, to clarify whether you are using a holding register or an input register.
Absolute Address	additional properties	Differs from **`Data Address`** only if using device Base Addresses. It is the sum of the **`Data Address`** value and the associated Base Address value (as configured in the parent Modbus device). This is the actual address that will be used when polling for this discrete data point from the actual Modbus device. The address of the polled data item is a combination of: • Address Format — either Modbus (default), Hex, or Decimal. • Address — numerical address, expressed in the selected **`Data Address`** format.
Data Source	read-only	Identifies where the data came from, such as "Point Poll."
Reg Type	drop-down	Selects the type of register.
Beginning Bit	a number from 0 to 15	Identifies the bit in the register where status information starts.
Number of Bits	number from 1–6	Defines the number of bits used for this enum point.
Out	read-only	Displays the current value of the proxy point including facets and status. The value depends on the type of control point. Facets define how the value displays, including the value's number of decimal places, engineering units, or text descriptors for Boolean/enum states. You can edit point facets to poll for additional properties, such as the native statusFlags

Property	Value	Description
		and/or priorityArray level. Status reports the current health and validity of the value. Status is specified by a combination of status flags, such as `fault`, `overridden`, `alarm`, and so on. If no status flag is set, status is considered normal and reports `{ok}`.

Proxy Ext (Client Numeric)

This is the proxy extension for either a ModbusClientNumericPoint (NumericPoint) or ModbusClientNumericWritable (NumericWritable). It contains information necessary to poll (read) an integer, long, float, or signed integer data value from a client **ModbusDevice**.

You access these properties by expanding **ModbusAsynNetwork** > **ModbusAsyncDevice** > **Points** in the Nav tree and double-clicking the ModbusClientNumericPoint or ModbusClientNumericWritable.

The ModbusClientNumericPoint or ModbusClientNumericWritable, has the following in addition to other Modbus client point Proxy Ext properties:

Property	Value	Description
Reg Type	drop-down list	Selects the type of register.
Data Type	drop-down list	Selects the location of the data in the slave device. Coils store on/off values. Registers store numeric values. Each coil or contact is 1 bit with an assigned address between 0000 and 270E. Each register is one word (16 bits, 2 bytes) as well as a data address between 0000 and 270E.

Proxy Ext (Client Numeric Bits)

This is the proxy extension for either a ModbusClientNumericBitsPoint (Numeric Bits Point) or ModbusClientNumericBitsWritable (Numeric Bits Writable).
It supports reading both Modbus holding register or input register values, and extracting bits specfied by the **`Beginning Bit`** and **`Number of Bits`** properties. The combination of these bits is the point's value, as a StatusNumeric. The writable variant writes the point's value into the raw register bits at the specified Beginning Bit relative position. Like other Modbus proxy points, both point-level or device-level polling is supported.

You access these properties by expanding **ModbusAsynNetwork** > **ModbusAsyncDevice** > **Points** in the Nav tree and double-clicking the ModbusClientNumericBitsPoint or ModbusClientNumericBitsWritable.

The ModbusClientNumericBitsPoint or ModbusClientNumericBitsWritable, has the following in addition to other Modbus client point Proxy Ext properties:

Property	Value	Description
Reg Type	drop-down list	Selects the type of register.
Beginning Bit	a number from 0 to 15	Identifies the bit in the register where status information starts.
Number of Bits	number from 1–16	Defines the number of bits used for this enum point.

Proxy Ext (Client Register Bit)

This is the proxy extension for either a ModbusClientRegisterBitPoint (BooleanPoint) or ModbusClientRegisterBitWritable (BooleanWritable).
It contains information necessary to poll (read) a single bit value from either an input register or holding register in client Modbus device.

You access these properties by expanding **ModbusAsynNetwork** > **ModbusAsyncDevice** > **Points** in the Nav tree and double-clicking the ModbusClientRegisterBitPoint or ModbusClientRegisterBitWritable.

The ModbusClientRegisterBitPoint or ModbusClientRegisterBitWritable, has the followingin addition to other Modbus client point Proxy Ext properties:

Property	Value	Description
Reg Type	drop-down list	Selects the type of register.
Bit Number	a number from 0 to 15	Defines the bit in the register that is associated with the current point.

Proxy Ext (Client String Point)

This is the proxy extension for a ModbusClientStringPoint (StringPoint). It contains information necessary to poll (read) a string data value from a client Modbus device.

You access these properties by expanding **ModbusAsynNetwork** > **ModbusAsyncDevice** > **Points** in the Nav tree and double-clicking the ModbusClientStringPoint.

The ModbusClientStringPoint, has the following in addition to other Modbus client point Proxy Ext properties:

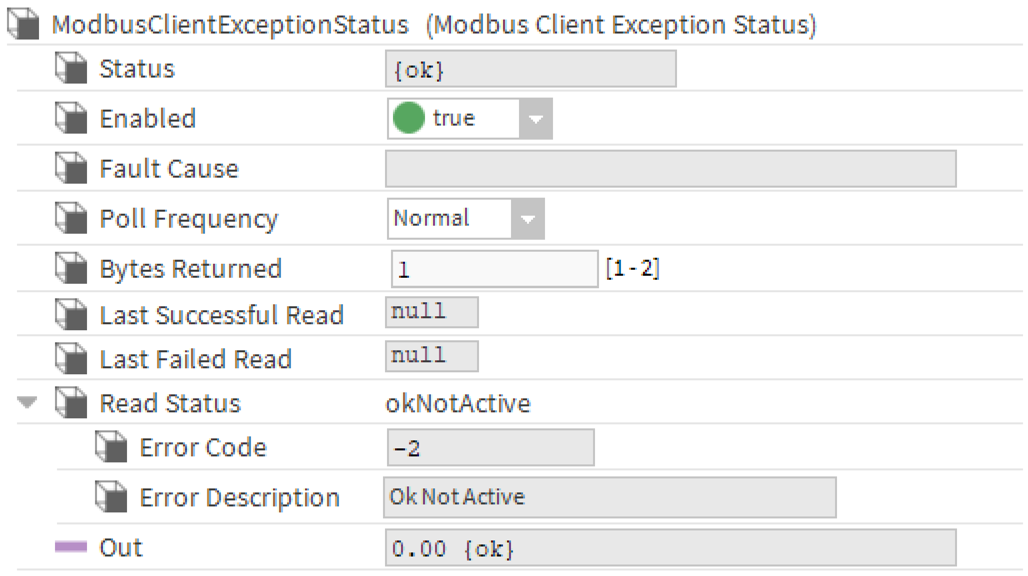
Property	Value	Description
Number Registers	number	Specifies the number of consecutive holding registers to read, starting with the specified **`Absolute Address`** (The number of registers should not exceed message limits of the target slave device). Each register produces two ASCII characters, using high-to-low byte order and standard ASCII encoding. For example, a register

Property	Value	Description
		with a value of `4A41` hex (19009 decimal) returns String characters JA, where: 4A h = J and 41 h = A.

ModbusClientExceptionStatus

ModbusClientExceptionStatus is a component used to read and expose Modbus function code 07 (Exception Status) data from a Modbus device.

Figure 13. ModbusClientExceptionStatus property sheet



ModbusClientExceptionStatus (Modbus Client Exception Status)

Status	{ok}
Enabled	● true
Fault Cause	
Poll Frequency	Normal
Bytes Returned	1 [1-2]
Last Successful Read	null
Last Failed Read	null
Read Status	okNotActive
Error Code	-2
Error Description	Ok Not Active
Out	0.00 {ok}

To use, you copy (drag) from the palette, and paste (drop) directly onto a client Modbus device —it must be a direct child of the **ModbusAsyncDevice**, **ModbusTcpDevice** or **ModbusTcpGatewayDevice**.

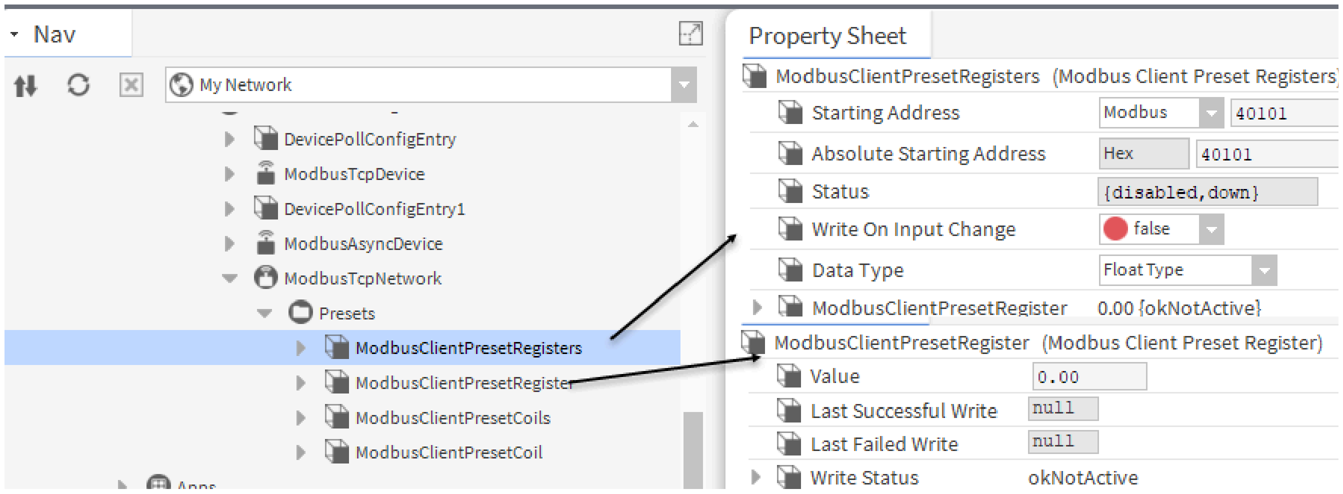
Property	Value	Description
Status	read-only	Reports the condition of the entity or process at last polling. `{ok}` indicates that the component is licensed and polling successfully. `{down}` indicates that the last check was unsuccessful, perhaps because of an incorrect property, or possibly loss of network connection. `{disabled}` indicates that the **`Enable`** property is set to `false`.

Property	Value	Description
		`{fault}` indicates another problem. Refer to **`Fault Cause`** for more information.
Enabled	`true` or `false` (defaults to `true`)	Activates (`true`) and deactivates (`false`) use of the object (network, device, point, component, table, schedule, descriptor, etc.).
Fault Cause	read-only	Indicates the reason why a system object (network, device, component, extension, etc.) is not working (in fault). This property is empty unless a fault exists.
Poll Frequency		Configures how frequently the system polls proxy points.
Bytes Returned	number (defaults to 1)	Specifies the exception status of the device whether it uses 1 or 2 bytes, and also the poll frequency to be used.
Last Successful Read	read-only date and time	Reports the last successful read.
Read Status	additional properties	Provides a numeric **`Error Code`** and **`Error Description`**, as well as timestamps of both the last successful and last failed read attempts.
Error Code	read-only number	Reports the number associated with a read or write error.
Error Description	read-only	Reports a short text description of the read or write error.
Out	additional properties	Exposes the exception status data as StatusBooleans on (Bit 0 to Bit 15) as well as a StatusNumeric Out slot.

ModbusClientPresetRegisters

In this preset container you specify the numerical data type for all child preset registers, and whether individual child preset register values are written to the Modbus slave upon any change, or only collectively when the Write action of the ModbusClientPresetRegisters container is invoked.

Figure 14. ModbusClientPresetRegisters properties



ModbusClientPresetRegisters (Modbus Client Preset Registers)

Starting Address	Modbus	40101
Absolute Starting Address	Hex	40101
Status	{disabled,down}	
Write On Input Change	false	
Data Type	Float Type	
ModbusClientPresetRegister	0.00 {okNotActive}	

ModbusClientPresetRegister (Modbus Client Preset Register)

Value	0.00
Last Successful Write	null
Last Failed Write	null
Write Status	okNotActive

You access these properties by expanding the `Presets` folder in the Nav tree and double-clicking the **`ModbusClientPresetRegisters`**.

Property	Value	Description
Starting Address	additional properties	Specifies the address of the first holding register to write (prior to any offset address change as a result of using device-level Base Address), as a combination of: • Address Format— either Hex (default), Decimal, or Modbus • Address — numerical address, expressed in the selected format.
Absolute Starting Address	read-only	Differs from Data Address only if using the device Base Addresses. It is the sum of the Data Address value and the associated Base Address value (as configured in the parent Modbus device). This is the actual address that uses when writing the first register's preset value.
Status	read-only	Displays the status of the container slot—can be fault if a previous write to a child preset register failed for some reason.
Write On Input Change	`true` or `false` (default)	Configures when to write a value

Property	Value	Description
		to the register. `true` writes a value when data coming in changes. `false` ignores value changes.
Data Type	drop-down	Selects the location of the data in the slave device. Coils store on/off values. Registers store numeric values. Each coil or contact is 1 bit with an assigned address between 0000 and 270E. Each register is one word (16 bits, 2 bytes) as well as a data address between 0000 and 270E.

Actions

Two right-click actions on a Modbus Client Preset Registers container are as follows:

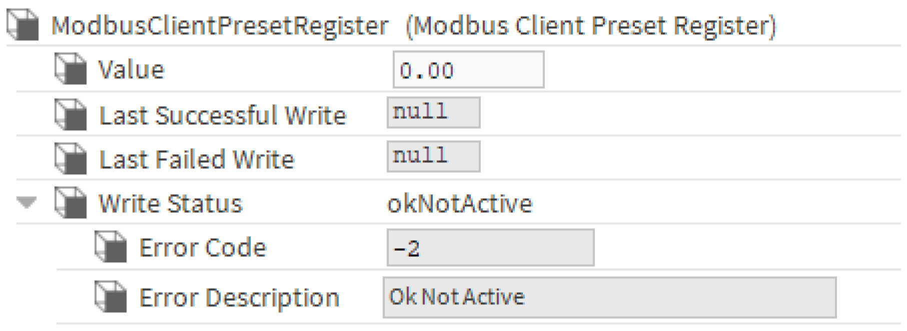
- Write
  To write all the preset register values currently in child ModbusClientPresetCoil components.
- Add Preset Register Value
  To add an additional child ModbusClientPresetRegister component, specifying its numerical value in the popup window. The component is added to the end of the existing slot order.

ModbusClientPresetRegister

ModbusClientPresetRegister is a child component of **ModbusClientPresetRegisters**, used to specify a preset numeric value to write to a holding register in a client Modbus device.

You can add any number of these register components under the ModbusClientPresetRegisters parent, where each specifies a numeric value to write. Data type can be integer, float, long, or signed integer, as specified by the configuration of the parent **ModbusClientPresetRegisters** container.

Figure 15. ModbusClientPresetRegister properties



ModbusClientPresetRegister (Modbus Client Preset Register)

Value	0.00
Last Successful Write	null
Last Failed Write	null
Write Status	okNotActive
Error Code	-2
Error Description	Ok Not Active

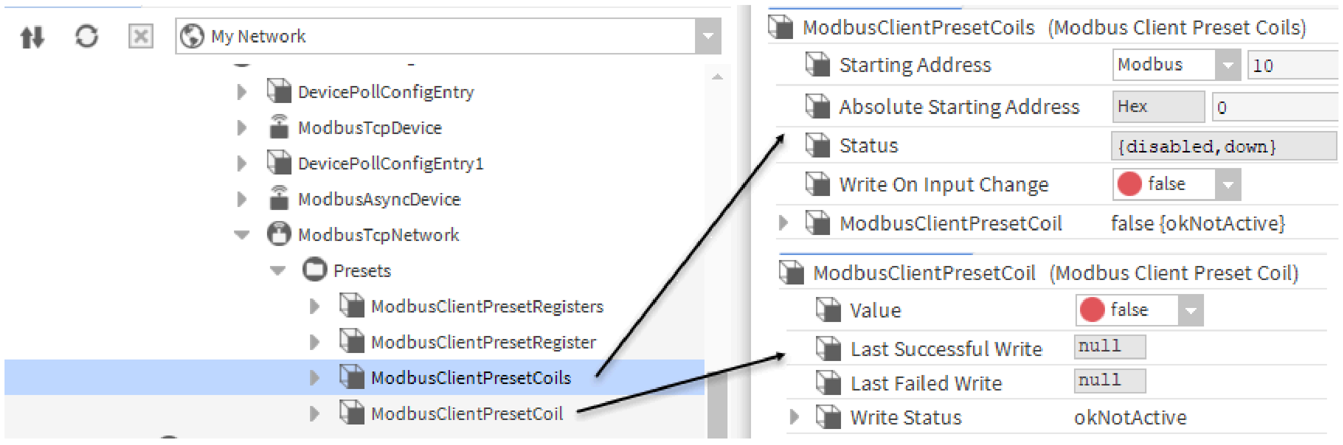
You access these properties by expanding **`ModbusClientPresetRegisters`** component in the Nav tree and double-clicking the **`ModbusClientPresetRegister`**.

Property	Value	Description
Value	number	Specifies a numeric value to write.
Last Successful Write	read-only date and time	Reports the last successful write.
Last Failed Write	read-only date and time	Reports the last failed write.
Write Status	read-only, writable or ok	Reports if the object is read-only or can be written to. `Read only` indicates the proxy extension cannot be written to. `Writable` or `ok` indicates that the object can be written to. For writable objects, this property indicates either that the last write occurred within the effective period or, if a write operation failed, it provides descriptive text.
Error Code	read-only number	Reports the number associated with a read or write error.
Error Description	text	Reports a short text description of the read or write error.

ModbusClientPresetCoils

This component contains one or more preset coil values (ModbusClientPresetCoil components). In this container, you specify a **`Starting Address`** for the first (topmost) child preset coil value. The driver sequentially addresses any additional child preset coil values relative to this slot. The driver always uses the **`Absolute Address`**for actual addressing, which is typically the same as the **`Starting Address`**, unless coils in the parent Modbus device are augmented with a base address.

Figure 16. ModbusClientPresetCoils properties



ModbusClientPresetCoils (Modbus Client Preset Coils)

Starting Address	Modbus 10
Absolute Starting Address	Hex 0
Status	{disabled,down}
Write On Input Change	false
ModbusClientPresetCoil	false {okNotActive}

ModbusClientPresetCoil (Modbus Client Preset Coil)

Value	false
Last Successful Write	null
Last Failed Write	null
Write Status	okNotActive

In this preset container you also specify whether individual child preset coil values are written to the Modbus slave upon any change, or only collectively when the Write action of the ModbusClientPresetCoils container is invoked.

You access these properties by expanding the `Presets` folder in the Nav tree and double-clicking the **`ModbusClientPresetCoils`**.

Property	Value	Description
Starting Address	additional properties	Specifies the address of the first holding register to write (prior to any offset address change as a result of using device-level Base Address), as a combination of: • Address Format— either Hex (default), Decimal, or Modbus • Address — numerical address, expressed in the selected format.
Absolute Starting Address	read-only	Differs from Data Address only if using the device Base Addresses. It is the sum of the Data Address value and the associated Base Address value (as configured in the parent Modbus device). This is the actual

Property	Value	Description
		address that uses when writing the first register's preset value.
Status	read-only	Displays the status of the container slot—can be fault if a previous write to a child preset register failed for some reason.
Write On Input Change	`true` or `false` (default)	Configures when to write a value to the register. `true` writes a value when data coming in changes. `false` ignores value changes.

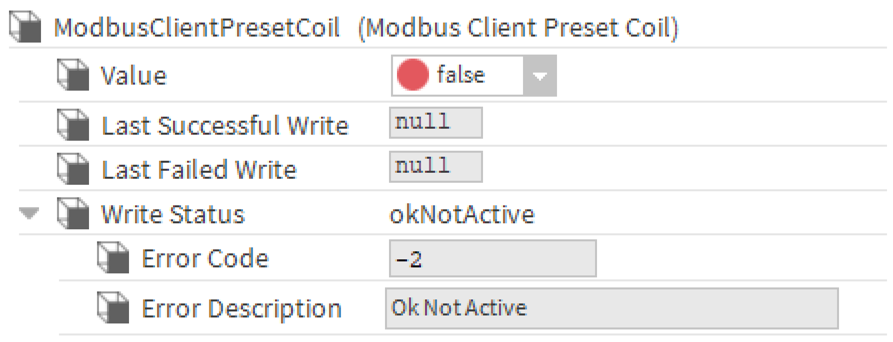
Actions

Two right-click actions on a Modbus Client Preset Coils container are as follows:

- Write
  To write all the preset coil values currently in child ModbusClientPresetCoil components.
- Add Preset Coil Value
  To add an additional child ModbusClientPresetCoil component, specifying its Boolean value in the popup window. The component is added to the end of the existing slot order.

ModbusClientPresetCoil

ModbusClientPresetCoil is a child component of **ModbusClientPresetCoils**. Use it to specify a single preset Modbus coil data value (`false` or `true`) to write to the parent client Modbus device. You can add any number of these preset coil components under the **ModbusClientPresetCoils** parent, where each specifies a Boolean value to write.



ModbusClientPresetCoil (Modbus Client Preset Coil)

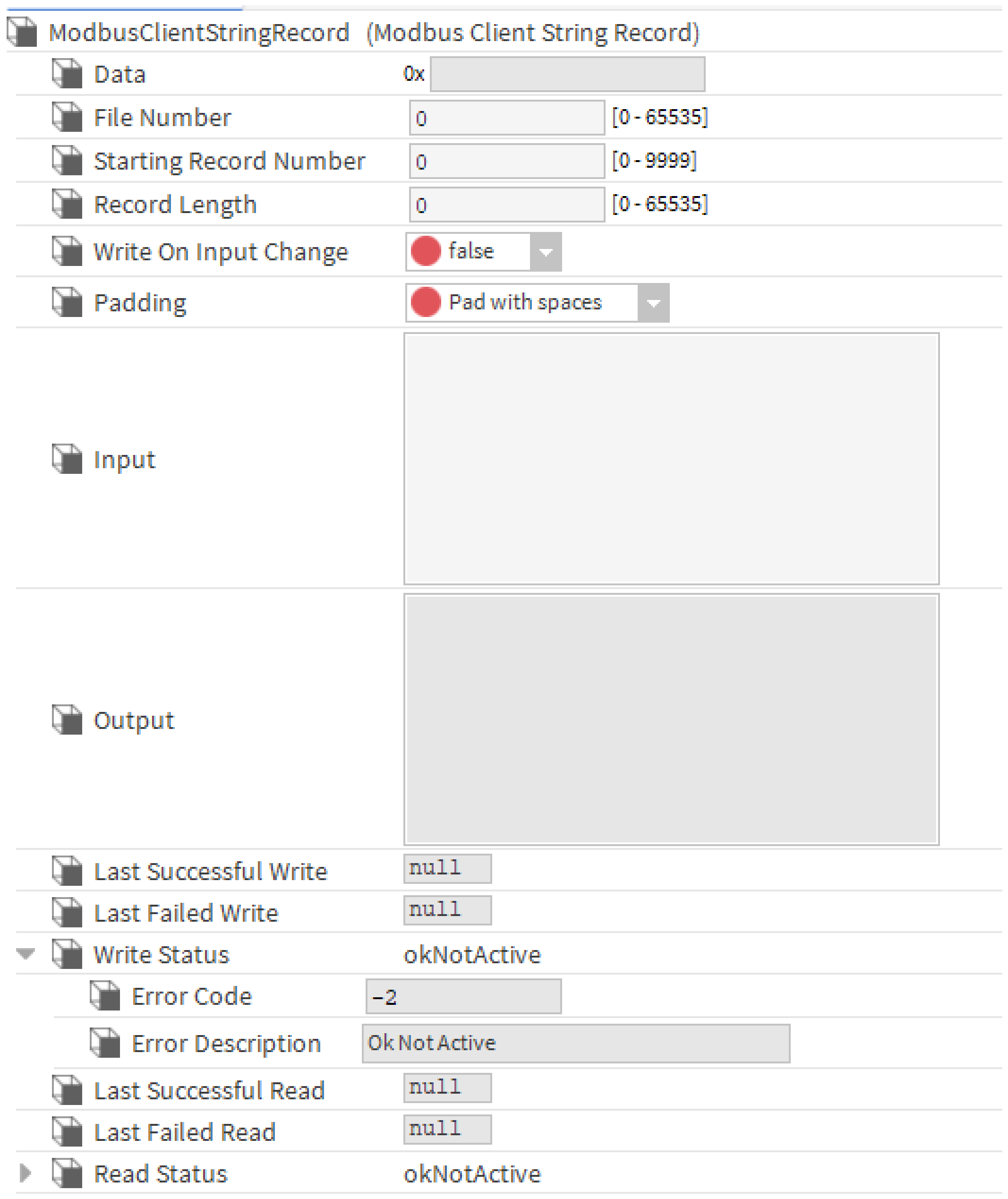
Value	● false
Last Successful Write	null
Last Failed Write	null
Write Status	okNotActive
Error Code	-2
Error Description	Ok Not Active

You access these properties by expanding the **ModbusClientPresetCoils** container in the Nav tree and double-clicking the **`ModbusClientPresetCoil`**.

Property	Value	Description
Value	`true` or `false` (default)	Controls if you can specify a value. `true` permits value specification. `false` prohibits value specification.
Last Successful Write	read-only date and time	Reports the last successful write.
Last Failed Write	read-only date and time	Reports the last failed write.
Write Status	addition properties	Reports if the proxy extension is read-only or can be written to.
Error Code	read-only	Reports the number associated with a read or write error.
Error Description	text	Reports a short text description of the read or write error.

ModbusClientStringRecord

This is a component for reading/writing Modbus file records (client side). The input and output is a string converted to/from a byte array. Writing occurs when the linkable write action is fired. Reading occurs when the linkable read action is fired.
As needed, you can copy this component from the **modbusAsync** or **modbusTcp** palette into a ModbusAsyncDevice, ModbusTcpDevice, or ModbusTcpGatewayDevice.



ModbusClientStringRecord	(Modbus Client String Record)	
Data	0x	
File Number	0	[0 - 65535]
Starting Record Number	0	[0 - 9999]
Record Length	0	[0 - 65535]
Write On Input Change	false	
Padding	Pad with spaces	
Input		
Output		
Last Successful Write	null	
Last Failed Write	null	
Write Status	okNotActive	
Error Code	-2	
Error Description	Ok Not Active	
Last Successful Read	null	
Last Failed Read	null	
Read Status	okNotActive	

Property	Value	Description
Data	read-only	Displays the value of the converted string as ASCII characters.
File Number	number (defaults to 0)	Identifies the source file for the data as a value from 0 to 65535.
Starting Record Number	number (defaults to 0)	Displays the record at which the converted string begins as a number fro 0 to 9999.

Property	Value	Description
Record Length	number (defaults to 0)	Displays the length of the converted record from 0 to 65535.
Write On Input Change	`true` or `false` (default)	Configures when to write a value to the register. `true` writes a value when data coming in changes. `false` ignores value changes.
Padding	drop-down list (defaults to Pad with spaces)	Indicates the type of characters used to fill out the record.
Input	text box	Displays the data coming in as a string from a byte array.
Output	text box	Displays the data going out as a string converted to a byte array.
Last Successful Write	read-only date and time	Reports the last successful write.
Last Failed Write	read-only date and time	Reports the last failed write.
Write Status	additional properties	Reports if the object is read-only or can be written to.
Last Successful Read	read-only date and time	Reports the last successful read.
Last Failed Read	read-only date and time	Reports the last failed read.
Read Status	additional properties	Reports a numerical Error Code (0-2), and a corresponding Error Description.

ModbusAsyncDeviceFolder

This is the ModbusAsync implementation of a folder under a ModbusAsyncNetwork. You can use these folders to organize ModbusAsyncDevices in the network.
Typically, you add such folders using the **New Folder** button in the **Modbus Async Device Manager** view of the network. Each device folder has its own device manager view. The ModbusAsyncDeviceFolder is also available in the **modbusAsync** palette.

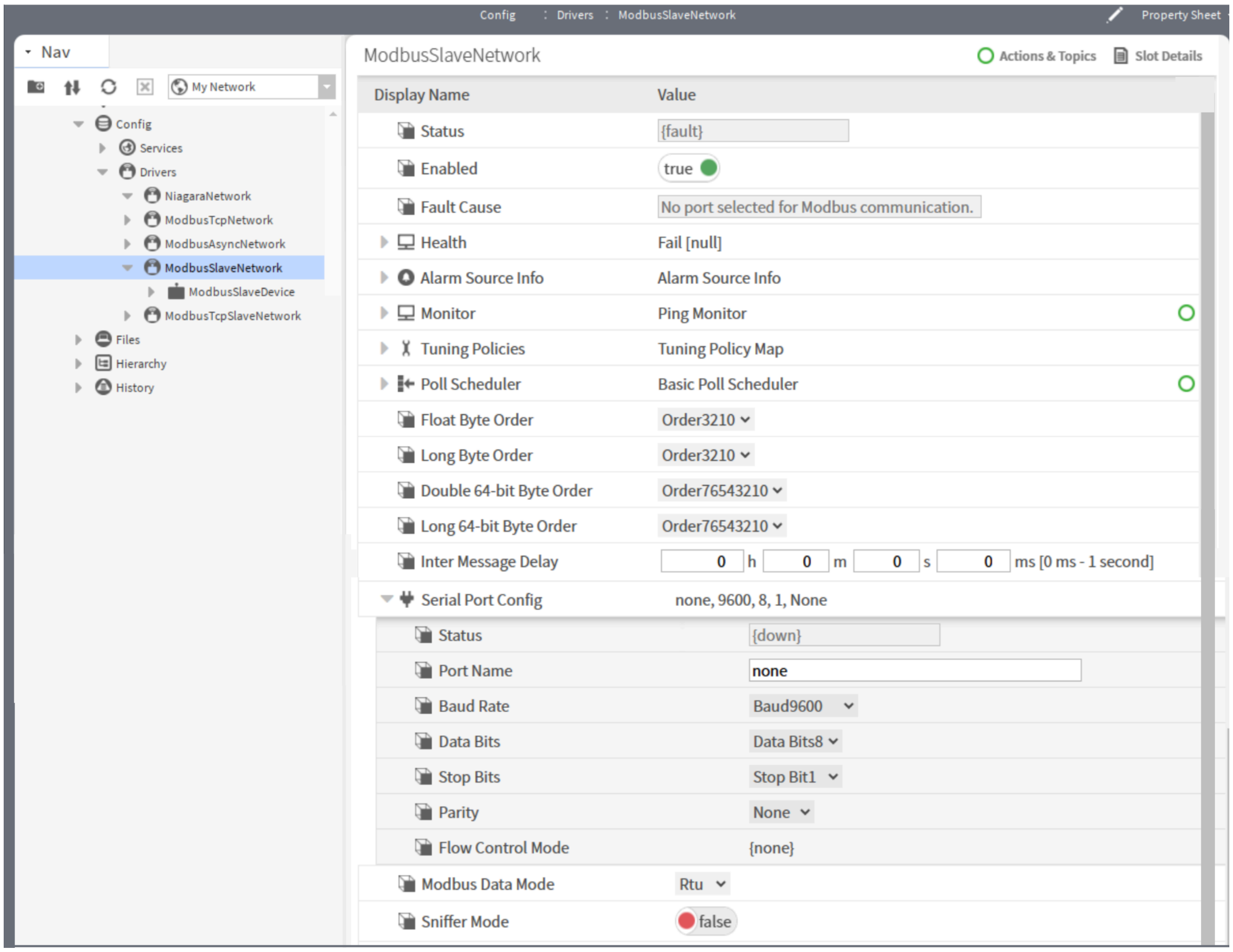
ModbusClientPointFolder

This is the Modbus client implementation of a folder under the **Points** container (ModbusClientPointDeviceExt) of a **ModbusAsyncDevice** and **ModbusTcpDevice**.
You typically add such folders using the **New Folder** button in the **Modbus Client Point Manager**. Each points folder also has its own Point Manager view.

ModbusSlaveNetwork

This component configures a Modbus slave network. A slave-type Modbus network allows the station to appear as one or more virtual Modbus slave devices, each providing some number of Modbus virtual data items. On the Modbus network the station simply waits for (and responds to) client requests from a master device. All proxy points are Modbus server types, where point polling monitors for external writes to some proxy points. Data exchange with the Modbus master occurs in this fashion with proxy points, and in rare cases with reads and writes to server file records (for string data).

Figure 17. ModbusSlaveNetwork property sheet



Config : Drivers : ModbusSlaveNetwork — Property Sheet

Nav — My Network
- Config
  - Services
  - Drivers
    - NiagaraNetwork
    - ModbusTcpNetwork
    - ModbusAsyncNetwork
    - ModbusSlaveNetwork
      - ModbusSlaveDevice
    - ModbusTcpSlaveNetwork
- Files
- Hierarchy
- History

ModbusSlaveNetwork — Actions & Topics — Slot Details

Display Name	Value
Status	{fault}
Enabled	true
Fault Cause	No port selected for Modbus communication.
Health	Fail [null]
Alarm Source Info	Alarm Source Info
Monitor	Ping Monitor
Tuning Policies	Tuning Policy Map
Poll Scheduler	Basic Poll Scheduler
Float Byte Order	Order3210
Long Byte Order	Order3210
Double 64-bit Byte Order	Order76543210
Long 64-bit Byte Order	Order76543210
Inter Message Delay	0 h 0 m 0 s 0 ms [0 ms - 1 second]
Serial Port Config	none, 9600, 8, 1, None
Status	{down}
Port Name	none
Baud Rate	Baud9600
Data Bits	Data Bits8
Stop Bits	Stop Bit1
Parity	None
Flow Control Mode	{none}
Modbus Data Mode	Rtu
Sniffer Mode	false

The ModbusSlaveNetwork has the standard collection of network components, such as for status, health, monitor, tuning policies, and poll scheduler. For more details, refer to the *Niagara Drivers Guide*.

You can access these properties by double-clicking **Driver** > **ModbusSlaveNetwork** in the Nav tree.

Property	Value	Description
Float Byte Order	drop-down list	Specifies the byte-order used to assemble or receive floating-

Property	Value	Description
		point (32-bit) values in messages. Choices reflect two alternate methods, where numerals 0, 1, 2, and 3 represent the least significant byte to most significant byte: • `Order3210` – Most significant byte first, or big-endian, it is the default. • `Order1032` – Bytes transmitted in a 1,0,3,2 order, or little–endian. • `Order0123` – Bytes transmitted in a 0,1,2,3 order, or little-endian.
Long Byte Order	drop-down list	Specifies the byte-order used to assemble or receive long integer (32-bit) values in messages. Choices reflect two alternate methods, where numerals 0, 1, 2, and 3 represent the least significant byte to most significant byte: • `Order3210` - Most significant byte first, or big-endian, it is the default. • `Order1032` - Bytes transmitted in a 1,0,3,2 order, or little-endian. • `Order0123` - Bytes transmitted in a 0,1,2,3 order, or little-endian. **NOTE:** Float or long values received in incorrect byte order may appear abnormally big, or not at all.
Double 64-bit Byte Order	drop-down list	Specifies the byte-order used to assemble or receive receives double-precision floating point (64-bit) values in messages. Choices reflect in 8 corresponding byte order options, where numerals 0, 1, 2, and 3 represent the least significant byte to most significant byte. When selecting the byte order options, it is required that the Float Byte

Property	Value	Description
		Order and Long Byte Order are both set to `Order0123` to implement this configuration effectively: • `Order76543210` – Most significant byte first, or big-endian order where the most significant byte is transmitted first (from 7 down to 0). • `Order 67452301` – Bytes transmitted in a order 6,7,4,5, etc or big-endian format involves swapping the bytes. • `Order54761032` – Bytes transmitted in a order 5, 4, 7, 6, etc or big-endian format but with each pair of bytes swapped (e.g., switching the first two bytes and the next two bytes). • `Order45670123` – Bytes are transmitted in order 4, 5, 6, 7, etc or big-endian with both words and bytes swapped. • `Order01234567` – Bytes are transmitted in order 0, 1, 2, 3, etc or little-endian (LE) arrangement where the bytes are transmitted in direct order from least significant to most significant (from 0 to 7). • `Order10325476` – Bytes are transmitted in order 1, 0, 3, 2, etc or little-endian format where the bytes are swapped during transmission. • `Order23016745` – Bytes are transmitted in order 2, 3, 0, 1, etc or little-endian format but with each pair of bytes swapped (e.g., switching the first two bytes and the next two bytes). • `Order32107654` – Bytes are transmitted in order 3, 2, 1, 0, etc or little-endian

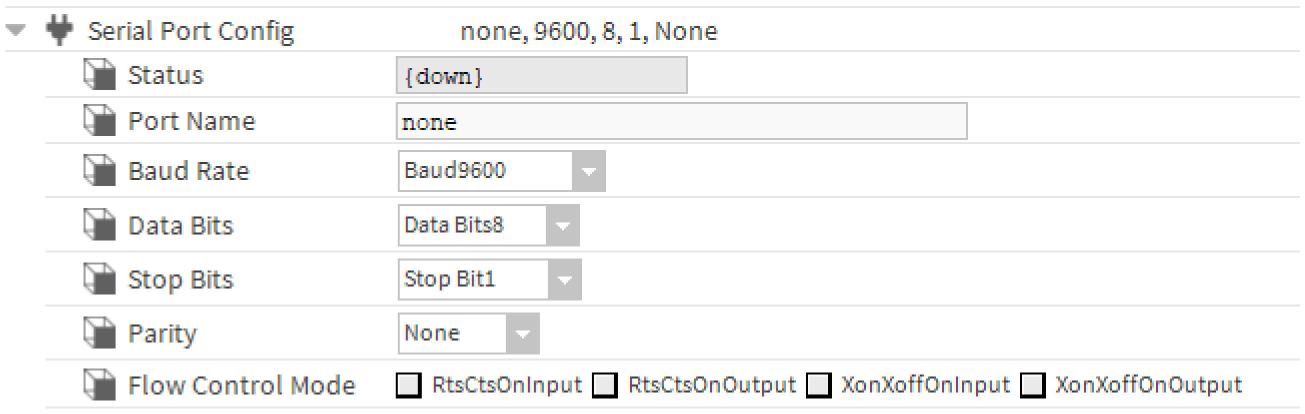
Property	Value	Description
		format, with both words and bytes swapped.
Long 64-bit Byte Order	drop-down list	Specifies the byte-order used to assemble or receive long integer (64-bit) values in messages. Choices reflect in 8 corresponding byte order options, where numerals 0, 1, 2, and 3 represent the least significant byte to most significant byte. When selecting the byte order options, it is required that the Float Byte Order and Long Byte Order are both set to `Order0123` to implement this configuration effectively: • `Order76543210` – Most significant byte first, or big-endian (BE) order where the most significant byte is transmitted first (from 7 down to 0). • `Order 67452301` – Bytes transmitted in a order 6,7,4,5, etc or big-endian format involves swapping the bytes. • `Order54761032` – Bytes transmitted in a order 5, 4, 7, 6, etc or big-endian format but with each pair of bytes swapped (e.g., switching the first two bytes and the next two bytes). • `Order45670123` – Bytes are transmitted in order 4, 5, 6, 7, etc. or big-endian with both words and bytes swapped. • `Order01234567` – Bytes are transmitted in order 0, 1, 2, 3, etc or little-endian (LE) arrangement where the bytes are transmitted in direct order from least significant to most significant (from 0 to 7). • `Order10325476` – Bytes are transmitted in order 1, 0,

Property	Value	Description
		3, 2, etc or little-endian format where the bytes are swapped during transmission. • `Order23016745` – Bytes are transmitted in order 2, 3, 0, 1, etc or little-endian format but with each pair of bytes swapped (e.g., switching the first two bytes and the next two bytes). • `Order32107654` – Bytes are transmitted in order 3, 2, 1, 0, etc or little-endian format, with both words and bytes swapped.
Serial Port Config	additional properties	Refer to Serial Port Config
Modbus Data Mode	drop-down list (defaults to Rtu)	Selects the type of serial network. `Rtu` (Remote Terminal Unit) `Ascii` (American Standard Code for Information Interchange)
Sniffer Mode	`true` or `false` (default)	Usually left at the default unless there is a particular reason to change it.

Serial Port Config

Specifies the serial port/communications setup required to talk to the attached serial Modbus master.

Figure 18. Serial Port Config properties



Serial Port Config	none, 9600, 8, 1, None
Status	{down}
Port Name	none
Baud Rate	Baud9600
Data Bits	Data Bits8
Stop Bits	Stop Bit1
Parity	None
Flow Control Mode	☐ RtsCtsOnInput ☐ RtsCtsOnOutput ☐ XonXoffOnInput ☐ XonXoffOnOutput

In addition to the standard property (Status), these properties are unique to serial port configuration:

Property	Value	Description
Port Name	text	Defines the communication port to use: `none`, `COM2` or `COM3`.
Baud Rate	drop-down list (defaults to `Baud9600`)	Defines communication speed in bits per second.
Data Bits	drop-down list (defaults to `Data Bits8`)	Defines the number of bits required to encode a character (a byte).
Stop Bits	drop-down list (defaults to `Stop Bit1`)	Defines the number of bits that indicate the end of a character.
Parity	drop-down list (defaults to `None`)	Defines the odd or even requirement of a transmitted byte of data for the purpose of error detection.
Flow Control Mode	check box	Using the selected Modbus protocol, manages the efficient transmission of data between two devices.

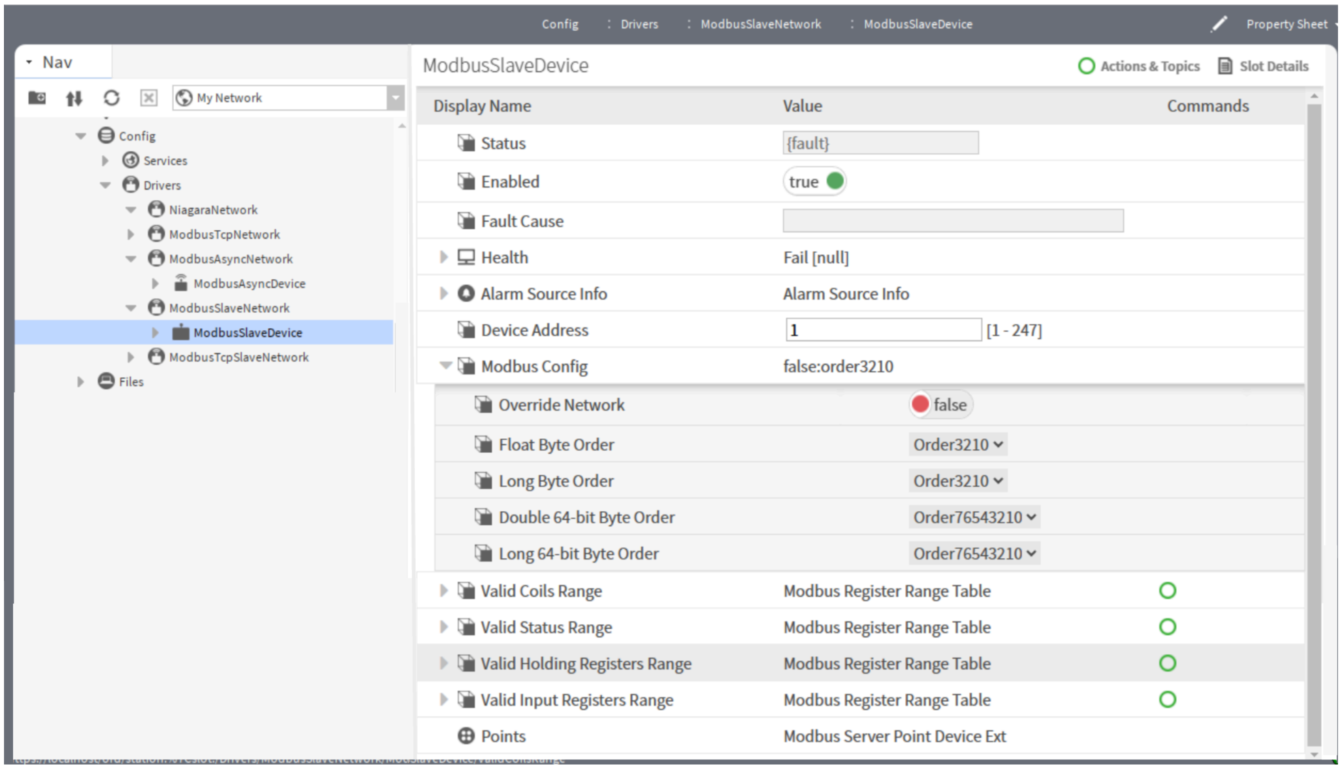
ModbusSlaveDeviceFolder

This is the ModbusSlave implementation of a folder under a ModbusSlaveNetwork. You can use these folders to organize ModbusSlaveDevices in the network.
Typically, you add such folders using the **New Folder** button in the **Modbus Slave Device Manager** view of the network. Each device folder has its own device manager view. The ModbusSlaveDeviceFolder is also available in the **modbusSlave** palette.

ModbusSlaveDevice

This component configures a Modbus slave device. You can specify many ranges of Modbus data items (coils, inputs, input registers, holding registers) in any or all ModbusSlaveDevice components.

Figure 19. ModbusSlaveDevice properties



Display Name	Value	Commands
Status	{fault}	
Enabled	true	
Fault Cause		
Health	Fail [null]	
Alarm Source Info	Alarm Source Info	
Device Address	1 [1 - 247]	
Modbus Config	false:order3210	
Override Network	false	
Float Byte Order	Order3210	
Long Byte Order	Order3210	
Double 64-bit Byte Order	Order76543210	
Long 64-bit Byte Order	Order76543210	
Valid Coils Range	Modbus Register Range Table	○
Valid Status Range	Modbus Register Range Table	○
Valid Holding Registers Range	Modbus Register Range Table	○
Valid Input Registers Range	Modbus Register Range Table	○
Points	Modbus Server Point Device Ext	

You access these properties by expanding **Drivers** > **ModbusSlaveNetwork** and double-clicking the **ModbusSlaveDevice** in the Nav tree.

In addition to the standard properties (Status, Enabled, Fault Cause, Health and Alarm Source Info), these properties support the Modbus async device:

Property	Value	Description
Device Address	number from 1 to 247	Defines the unique number that identifies the current device object on the network.
Modbus Config	additional properties	Refer to Modbus Config .
Valid Coils Range	additional properties	Refer to ValidCoilsRange .
Valid Status Range	additional properties	Refer to ValidCoilsRange for properties.
Valid Holding Registers Range	additional properties	Refer to ValidCoilsRange for properties.
Valid Input Registers Range	additional properties	Refer to ValidCoilsRange for properties.

Property	Value	Description
Points	additional properties	Provides a container for proxy points.

Modbus Config

Property	Value	Description
Override Network	`true` or `false` (default)	Determines which values to use for these properties: **`Float Byte Order`**, **`Long Byte Order`** and **`Use Force Multiple Coil`**. `false` selects the network-level values as configured by the ModbusAsyncNetwork component. `true` selects the values defined by the **Modbus Config** container slot.
Float Byte Order	drop-down list	Specifies the byte-order used to assemble or receive floating-point (32-bit) values in messages. Choices reflect two alternate methods, where numerals 0, 1, 2, and 3 represent the least significant byte to most significant byte: • `Order3210` – Most significant byte first, or big-endian, it is the default. • `Order1032` – Bytes transmitted in a 1,0,3,2 order, or little–endian. • `Order0123` – Bytes transmitted in a 0,1,2,3 order, or little-endian.
Long Byte Order	drop-down list	Specifies the byte-order used to assemble or receive long integer (32-bit) values in messages. Choices reflect two alternate methods, where numerals 0, 1, 2, and 3 represent the least significant byte to most significant byte: • `Order3210` - Most significant byte first, or big-endian, it is the default. • `Order1032` - Bytes

Property	Value	Description
		transmitted in a 1,0,3,2 order, or little-endian. • `Order0123` - Bytes transmitted in a 0,1,2,3 order, or little-endian. **NOTE:** Float or long values received in incorrect byte order may appear abnormally big, or not at all.
Double 64-bit Byte Order	drop-down list	Specifies the byte-order used to assemble or receive receives double-precision floating point (64-bit) values in messages. Choices reflect in 8 corresponding byte order options, where numerals 0, 1, 2, and 3 represent the least significant byte to most significant byte. When selecting the byte order options, it is required that the Float Byte Order and Long Byte Order are both set to `Order0123` to implement this configuration effectively: • `Order76543210` – Most significant byte first, or big-endian order where the most significant byte is transmitted first (from 7 down to 0). • `Order 67452301` – Bytes transmitted in a order 6,7,4,5, etc or big-endian format involves swapping the bytes. • `Order54761032` – Bytes transmitted in a order 5, 4, 7, 6, etc or big-endian format but with each pair of bytes swapped (e.g., switching the first two bytes and the next two bytes). • `Order45670123` – Bytes are transmitted in order 4, 5, 6, 7, etc or big-endian with both words and bytes swapped. • `Order01234567` – Bytes

Property	Value	Description
		are transmitted in order 0, 1, 2, 3, etc or little-endian (LE) arrangement where the bytes are transmitted in direct order from least significant to most significant (from 0 to 7). • `Order10325476` – Bytes are transmitted in order 1, 0, 3, 2, etc or little-endian format where the bytes are swapped during transmission. • `Order23016745` – Bytes are transmitted in order 2, 3, 0, 1, etc or little-endian format but with each pair of bytes swapped (e.g., switching the first two bytes and the next two bytes). • `Order32107654` – Bytes are transmitted in order 3, 2, 1, 0, etc or little-endian format, with both words and bytes swapped.
Long 64-bit Byte Order	drop-down list	Specifies the byte-order used to assemble or receive long integer (64-bit) values in messages. Choices reflect in 8 corresponding byte order options, where numerals 0, 1, 2, and 3 represent the least significant byte to most significant byte. When selecting the byte order options, it is required that the Float Byte Order and Long Byte Order are both set to `Order0123` to implement this configuration effectively: • `Order76543210` – Most significant byte first, or big-endian (BE) order where the most significant byte is transmitted first (from 7 down to 0). • `Order 67452301` – Bytes transmitted in a order 6,7,4,5, etc or big-endian format involves swapping

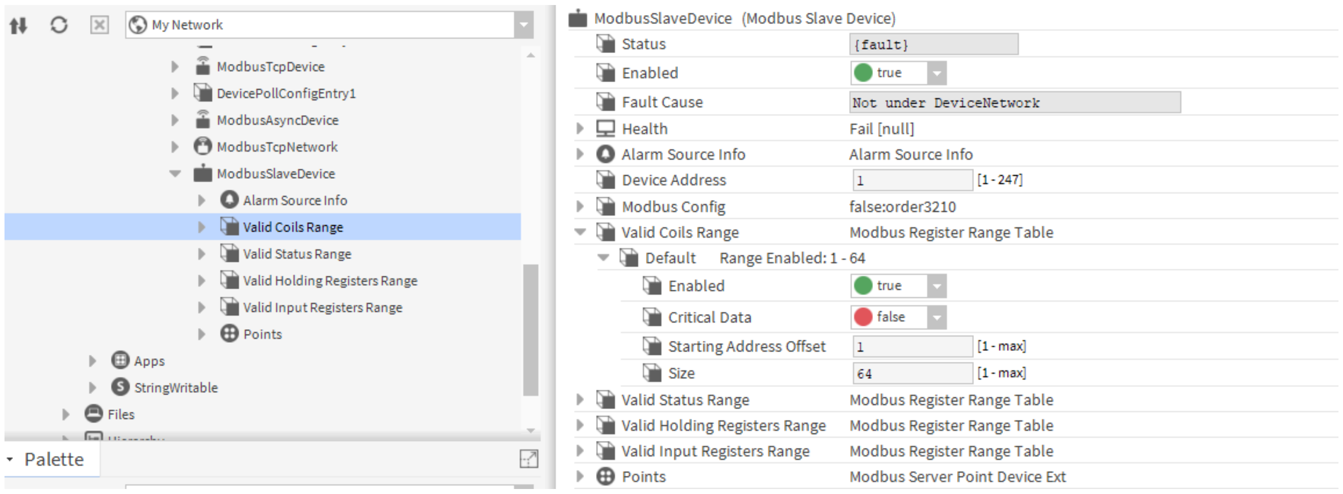
Property	Value	Description
		the bytes. • `Order54761032` – Bytes transmitted in a order 5, 4, 7, 6, etc or big-endian format but with each pair of bytes swapped (e.g., switching the first two bytes and the next two bytes). • `Order45670123` – Bytes are transmitted in order 4, 5, 6, 7, etc. or big-endian with both words and bytes swapped. • `Order01234567` – Bytes are transmitted in order 0, 1, 2, 3, etc or little-endian (LE) arrangement where the bytes are transmitted in direct order from least significant to most significant (from 0 to 7). • `Order10325476` – Bytes are transmitted in order 1, 0, 3, 2, etc or little-endian format where the bytes are swapped during transmission. • `Order23016745` – Bytes are transmitted in order 2, 3, 0, 1, etc or little-endian format but with each pair of bytes swapped (e.g., switching the first two bytes and the next two bytes). • `Order32107654` – Bytes are transmitted in order 3, 2, 1, 0, etc or little-endian format, with both words and bytes swapped.

ValidCoilsRange

This component is one of the frozen container slots available on a Modbus server (slave) device.

Queries received by the device must be for data items within the defined (and enabled) valid address ranges. Otherwise the station returns an exception response. In addition, server Modbus proxy points under the device must be configured to fall within these address ranges, or else they will have a fault status.

Figure 20. Modbus Register Range Tables of Modbus server device



My Network
- ModbusTcpDevice
- DevicePollConfigEntry1
- ModbusAsyncDevice
- ModbusTcpNetwork
- ModbusSlaveDevice
  - Alarm Source Info
  - Valid Coils Range
  - Valid Status Range
  - Valid Holding Registers Range
  - Valid Input Registers Range
  - Points
- Apps
- StringWritable
- Files

Palette

ModbusSlaveDevice (Modbus Slave Device)

Status	{fault}
Enabled	true
Fault Cause	Not under DeviceNetwork
Health	Fail [null]
Alarm Source Info	Alarm Source Info
Device Address	1 [1 - 247]
Modbus Config	false:order3210
Valid Coils Range	Modbus Register Range Table
Default	Range Enabled: 1 - 64
Enabled	true
Critical Data	false
Starting Address Offset	1 [1 - max]
Size	64 [1 - max]
Valid Status Range	Modbus Register Range Table
Valid Holding Registers Range	Modbus Register Range Table
Valid Input Registers Range	Modbus Register Range Table
Points	Modbus Server Point Device Ext

You can see these range components under the device when expanded in the Nav tree, as well as in the device's property sheet.

By default, as copied from the modbusSlave or modbusTcpSlave palette, a Modbus server device has a single Default-named **`Register Range Entry`** in each valid range container, with each having an enabled range of from 1 to 64, for **`Valid Coils Range`**. The same default applies to the **`Valid Holding Registers Range`**, and so on.

Property	Value	Description
Enabled	`true` (default) or `false`	`false` sets any associated server proxy points to fault status causing any queries received to this address range to return an exception response.
Starting Address Offset	number	Defines the first register in the specified range.
Size	number	Defines the number of registers in the range.

ModbusRegisterRangeEntry

This component configures the valid (usable) Modbus data items by specifying an address range starting from the offset and ranging through the size.
It is a child of a ModbusRegisterRangeTable (Valid Coils Range, Valid Status Range, Valid Holding Registers, Valid Input Registers) slot of a server Modbus device.

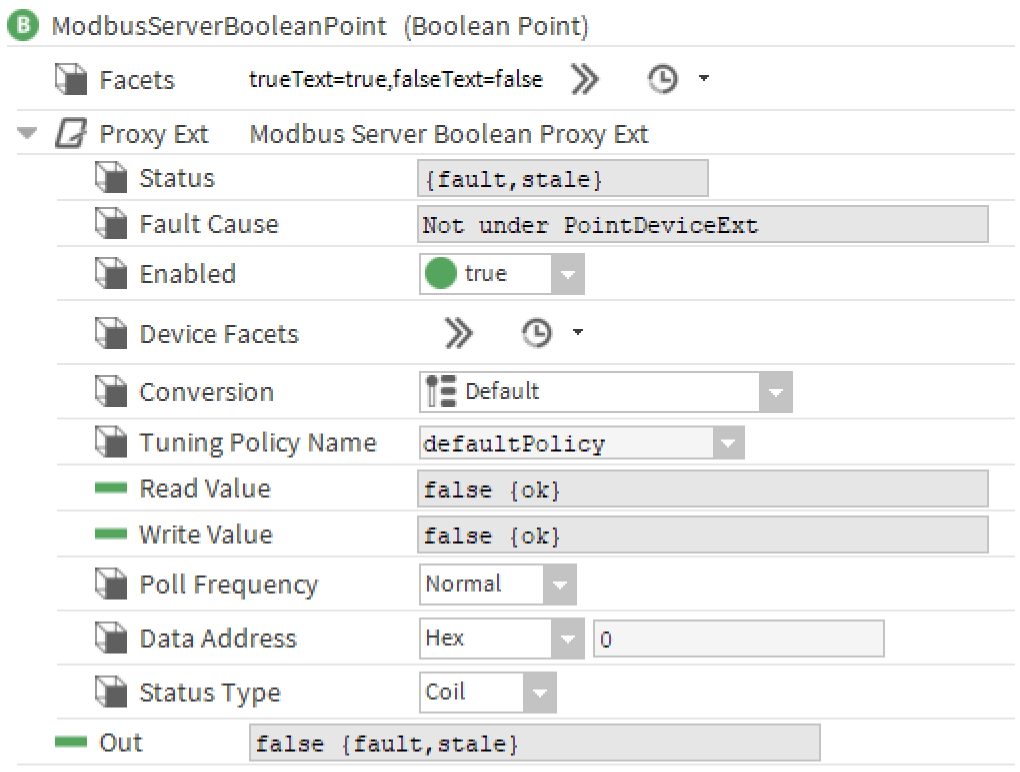
Points (Server)

This **Points** extension is the Modbus server implementation of PointDeviceExt, and is a frozen extension under every ModbusSlaveDevice and ModbusTcpSlaveDevice.
Its primary view is the **Modbus Server Point Manager**.

Proxy Ext (Server Boolean)

This is the proxy extension for either a ModbusServerBooleanPoint (BooleanPoint) or ModbusServerBooleanWritable (BooleanWritable). It contains information necessary information to poll (read) a status data value from a server Modbus device.

Figure 21. Proxy Ext (Server Boolean) Properties


ModbusServerBooleanPoint (Boolean Point)

Facets	trueText=true,falseText=false
Proxy Ext	Modbus Server Boolean Proxy Ext
Status	{fault,stale}
Fault Cause	Not under PointDeviceExt
Enabled	true
Device Facets	
Conversion	Default
Tuning Policy Name	defaultPolicy
Read Value	false {ok}
Write Value	false {ok}
Poll Frequency	Normal
Data Address	Hex 0
Status Type	Coil
Out	false {fault,stale}

You access these properties by expanding **ModbusSlaveNetwork** > **ModbusSlaveDevice** > **Points** in the Nav tree and double-clicking the ModbusServerBooleanPoint or ModbusServerBooleanWritable.

In addition to the standard properties (Status, Fault Cause and Enabled), these properties are unique to the Client Boolean proxy extension.

Property	Value	Description
Device Facets	Config Facets window	Determine how values are formatted for display depending on the context and the type of data. Examples include engineering units and decimal precision for numeric types, and descriptive value (state) text for boolean and enum types.

Property	Value	Description
		With the exception of proxy points (with possible defined device facets), point facets do not affect how the framework processes the point's value. Besides control points, various other components have facets too. For example, many kitControl and schedule components have facets. Details about point facets apply to these components too, unless especially noted. You access facets by clicking an **Edit** button or a chevron >>. Both open an **Edit Facets** window.
Conversion	drop-down list	Selects the units to use when converting values from the device facets to point facets. `Default` automatically converts similar units (such as Fahrenheit to Celsius) within the proxy point. **NOTE:** In most cases, the standard `Default` is best. `Linear` applies to voltage input, resistive input and voltage output writable points. Works with linear-acting devices. You use the Scale and Offset properties to convert the output value to a unit other than that defined by device facets. `Linear With Unit` is an extension to the existing linear conversion property. This specifies whether the unit conversion should occur on "Device Value" or "Proxy Value". The new linear with unit convertor, will have a property to indicate whether the unit conversion should take place before or after the scale/offset conversion.

Property	Value	Description
		`Reverse Polarity` applies only to Boolean input and relay output writable points. Reverses the logic of the hardware binary input or output. `500 Ohm Shunt` applies to voltage input points only. It reads a 4-to-20mA sensor, where the Ui input requires a 500 ohm resistor wired across (shunting) the input terminals. `Tabular Thermistor` applies to only a Thermistor input point and involves a custom resistance-to-temperature value response curve for Type 3 Thermistor temperature sensors. `Thermistor Type 3` applies to an Thermistor Input point, where this selection provides a "built-in" input resistance-to-temperature value response curve for Type 3 Thermistor temperature sensors. `Generic Tabular` applies to non-linear support for devices other than for thermistor temperature sensors with units in temperature. Generic Tabular uses a lookup table method similar to the "Thermistor Tabular" conversion, but without predefined output units.
Turning Policy Name	drop-down list (defaults to **`Default Policy`**)	Specifies the tuning policy to use for the proxy point.
Read Value	read-only	Reports the value read by the driver from the device and formatted based on device facets. This value agrees with point facets.
Write Value	read-only	Displays the last value written using device facets.
Poll Frequency	drop-down list	Configures how frequently the system polls proxy points.

Property	Value	Description

Data Address	drop-down list (defaults to `Modbus`)	Specifies the address of the polled data item (prior to any offset address change as a result of using device-level Base Address), as a combination of the address format and the numerical address expressed in the selected format. The formats are:
		
		`Modbus`
		
		`Hex`
		
		`Decimal`
		
		For example, the following are all equivalent addresses:
		• Modbus, 40012
		• Hex, 0B
		• Decimal, 11
		.
		
		NOTE: If you use the `Hex` or `Decimal` format for most read-only points you need to specify the **`Reg Type`** property, to clarify whether you are using a holding register or an input register.

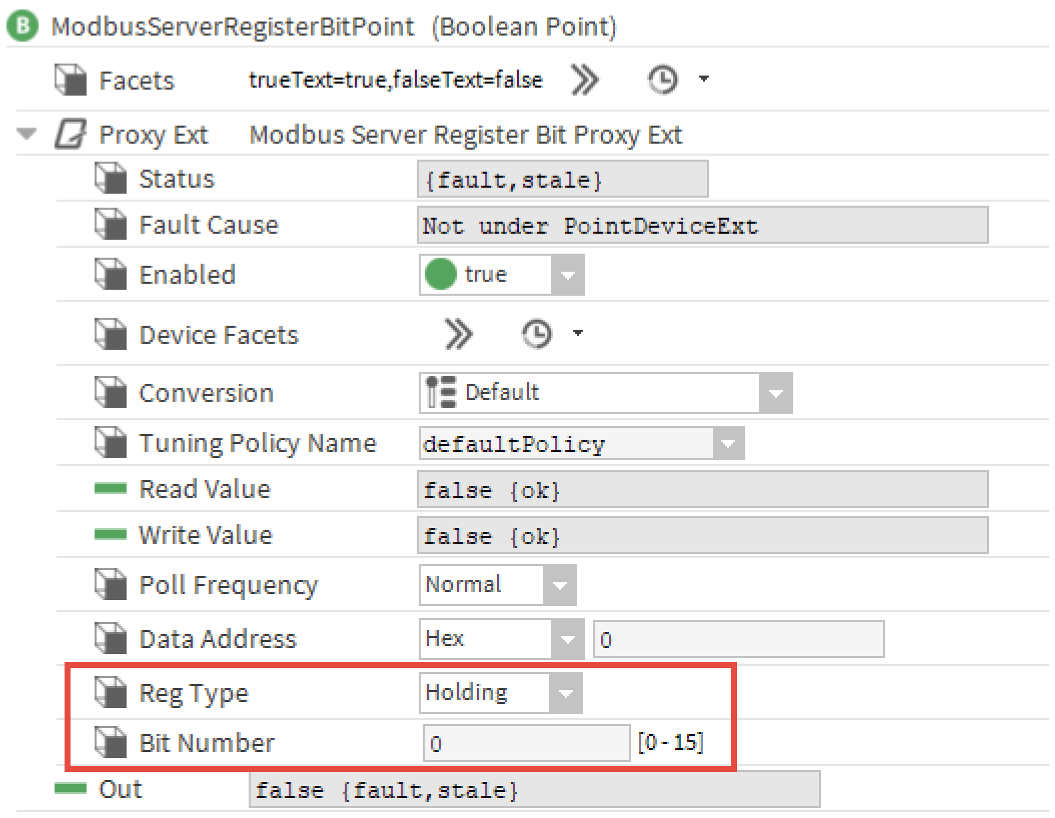
Data Source	read-only	Identifies where the data came from, such as “Point Poll.”
Status Type	drop-down list	Selects between `Coil` or `Input` to define the type to read. `Coil` is the only valid option (the master cannot write to Modbus inputs). Selection is only necessary if the **`Data Address`** format is set to `Hex` or `Decimal`. The Modbus **`Address Format`**, if used, automatically sets this property value.
Out	read-only	Displays the current value of the proxy point including facets and status. The value depends on the type of control point. Facets define how the value displays, including the value’s

Property	Value	Description
		number of decimal places, engineering units, or text descriptors for Boolean/enum states. You can edit point facets to poll for additional properties, such as the native statusFlags and/or priorityArray level. Status reports the current health and validity of the value. Status is specified by a combination of status flags, such as `fault`, `overridden`, `alarm`, and so on. If no status flag is set, status is considered normal and reports `{ok}`.

Proxy Ext (Server Numeric)

This is the proxy extension for either a ModbusServerNumericPoint (NumericPoint) or ModbusServerNumericWritable (NumericWritable). It contains information necessary to poll (read) an integer, long, float, or signed integer data value from a server .

Figure 22. Proxy Ext (Server Numeric) properties



ModbusServerRegisterBitPoint (Boolean Point)

Property	Value
Facets	trueText=true,falseText=false
Proxy Ext	Modbus Server Register Bit Proxy Ext
Status	{fault,stale}
Fault Cause	Not under PointDeviceExt
Enabled	true
Device Facets	
Conversion	Default
Tuning Policy Name	defaultPolicy
Read Value	false {ok}
Write Value	false {ok}
Poll Frequency	Normal
Data Address	Hex 0
Reg Type	Holding
Bit Number	0 [0 - 15]
Out	false {fault,stale}

You access these properties by expanding **ModbusSlaveNetwork** > **ModbusSlaveDevice** > **Points** in the Nav tree and double-clicking the ModbusServerNumericPoint or ModbusServerNumericWritable.

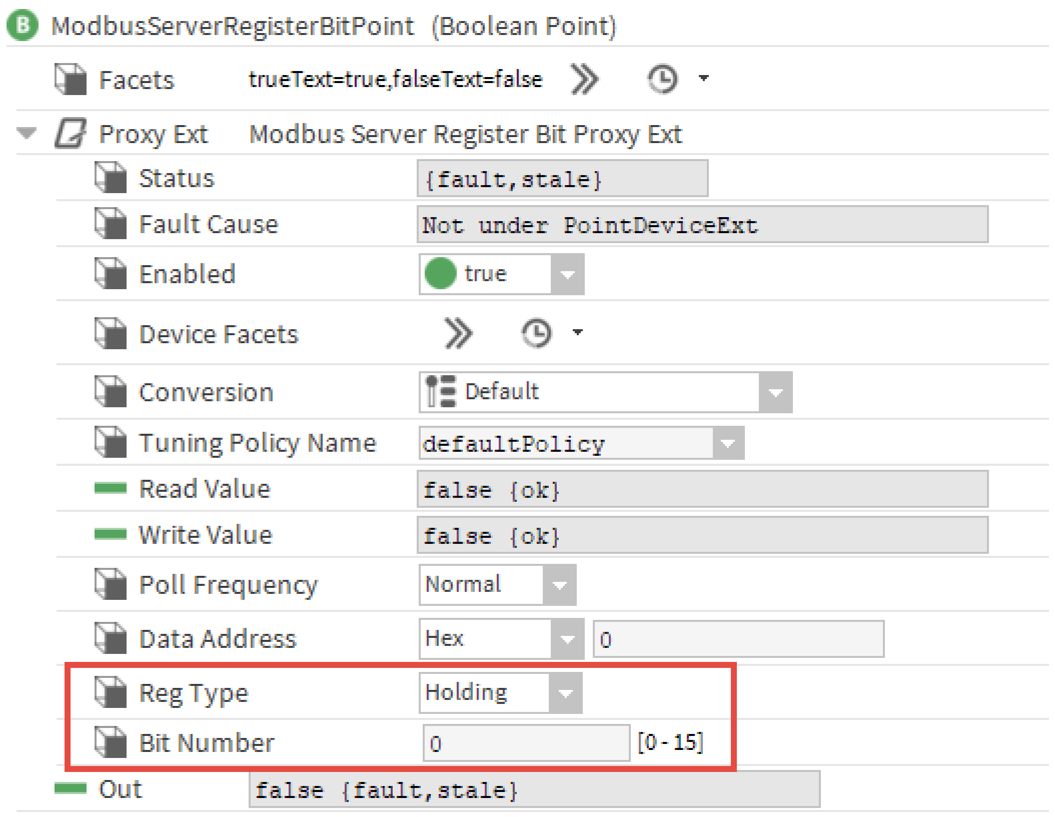
The ModbusServerNumericPoint or ModbusServerNumericWritable, has the following in addition to other Modbus serverpoint Proxy Ext properties:

Property	Value	Description
Reg Type	drop-down list	Selects the type of register.
Data Type	drop-down list	Selects the location of the data in the slave device. Coils store on/off values. Registers store numeric values. Each coil or contact is 1 bit with an assigned address between 0000 and 270E. Each register is one word (16 bits, 2 bytes) as well as a data address between 0000 and 270E.

Proxy Ext (Server Register Bits)

This is the proxy extension for either a ModbusServerRegisterBitPoint (BooleanPoint) or ModbusServerRegisterBitWritable (BooleanWritable).

Figure 23. Proxy Ext (Server Register Bits) properties


ModbusServerRegisterBitPoint (Boolean Point)

Facets	trueText=true,falseText=false
Proxy Ext	Modbus Server Register Bit Proxy Ext
Status	{fault,stale}
Fault Cause	Not under PointDeviceExt
Enabled	true
Device Facets	
Conversion	Default
Tuning Policy Name	defaultPolicy
Read Value	false {ok}
Write Value	false {ok}
Poll Frequency	Normal
Data Address	Hex 0
Reg Type	Holding
Bit Number	0 [0 - 15]
Out	false {fault,stale}

It contains information necessary to poll (read) a single bit value from either an input register or holding register in client Modbus device.

You access these properties by expanding **ModbusSlaveNetwork** > **ModbusSlaveDevice** > **Points** in the Nav tree and double-clicking the ModbusServerRegisterBitPoint or ModbusServerRegisterBitWritable.

The ModbusServerRegisterBitPoint or ModbusServerRegisterBitWritable, has the following in addition to other Modbus client point Proxy Ext properties:

Property	Value	Description
Reg Type	drop-down	Selects the type of register.
Bit Number	a number from 0 to 15	Defines the bit in the register that is associated with the current point.

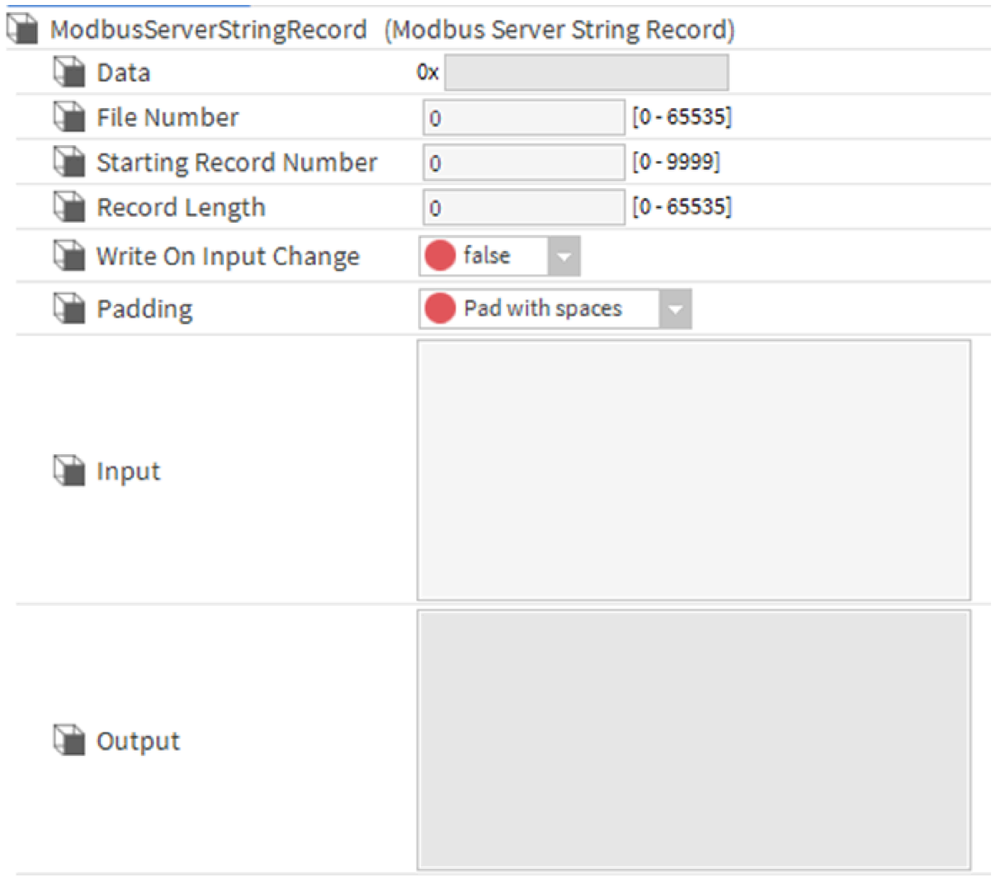
ModbusServerPointFolder

This is the Modbus server implementation of a folder under the **Points** container ModbusServerPointFolder of a ModbusSlaveDevice and ModbusTcpSlaveDevice.
You typically add such folders using the **New Folder** button in the **Modbus Server Point Manager**. Each points folder also has its own Point Manager view.

ModbusServerStringRecord

A Modbus Server String Record allows writing Modbus file records (server side support for Modbus function codes 20 and 21). The input and output is a string converted to/from a byte array. Writing occurs when the linkable write action is fired.

Figure 24. ModbusServerStringRecord properties



ModbusServerStringRecord (Modbus Server String Record)

Data	0x
File Number	0 [0 - 65535]
Starting Record Number	0 [0 - 9999]
Record Length	0 [0 - 65535]
Write On Input Change	false
Padding	Pad with spaces
Input	
Output	

This component allows you to locally set the value of a string file record, and also accepts read and write messages for the specified file record. It also converts the data to ASCII characters to display as a string.

To use, copy from the **modbusSlave** or **modbusTcpSlave** palette and place anywhere under the Modbus server device

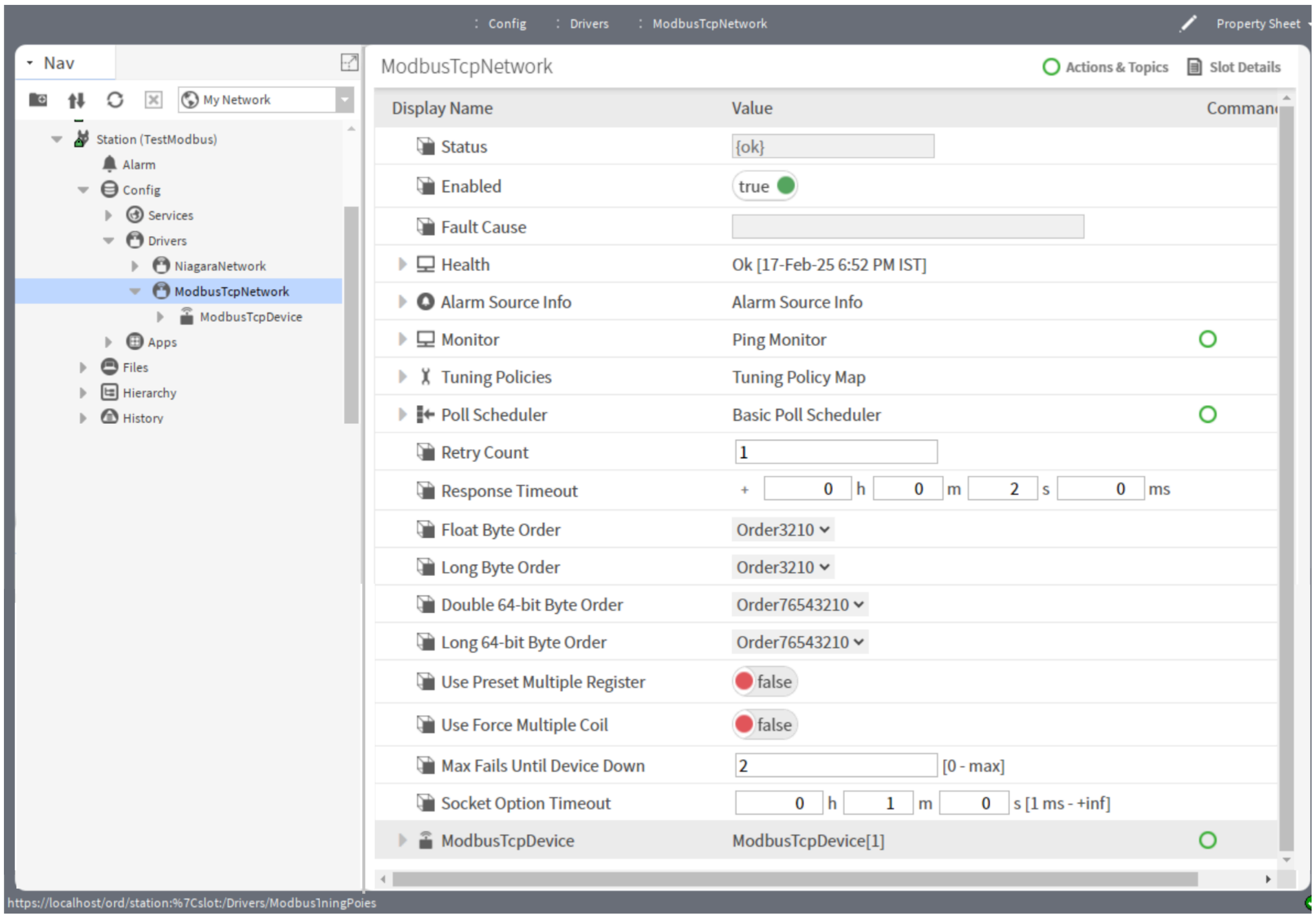
NOTE: it is not a proxy point—if you put under the device's **Points** container it will not be visible in any **Modbus Server Point Manager** view.

Property	Value	Description
Data	read-only	Displays the value of the converted string as ASCII characters.
File Number	number (defaults to 0)	Identifies the source file for the data as a value from 0 to 65535.
Starting Record Number	number (defaults to 0)	Displays the record at which the converted string begins as a number fro 0 to 9999.
Record Length	number (defaults to 0)	Displays the length of the converted record from 0 to 65535.
Write On Input Change	`true` or `false`	Configures when to write a value to the register. `true` writes a value when data coming in changes. `false` ignores value changes.
Padding	drop-down list (defaults to Pad with spaces)	Indicates the type of characters used to fill out the record.
Input	text box	Displays the data coming in as a string from a byte array.
Output	text box	Displays the data going out as a string converted to a byte array.

ModbusTcpNetwork

This component manages a TCP/IP network. In the property sheet of the ModbusTcpNetwork, you review the default global values for Modbus device data (the network automatically binds to the local TCP/IP address of the controller).

Figure 25. ModbusTcpNetwork property sheet



: Config : Drivers : ModbusTcpNetwork Property Sheet

Nav

My Network

- Station (TestModbus)
  - Alarm
  - Config
    - Services
    - Drivers
      - NiagaraNetwork
      - ModbusTcpNetwork
        - ModbusTcpDevice
    - Apps
  - Files
  - Hierarchy
  - History

ModbusTcpNetwork — Actions & Topics — Slot Details

Display Name	Value	Command
Status	{ok}	
Enabled	true	
Fault Cause		
Health	Ok [17-Feb-25 6:52 PM IST]	
Alarm Source Info	Alarm Source Info	
Monitor	Ping Monitor	
Tuning Policies	Tuning Policy Map	
Poll Scheduler	Basic Poll Scheduler	
Retry Count	1	
Response Timeout	+ 0 h 0 m 2 s 0 ms	
Float Byte Order	Order3210	
Long Byte Order	Order3210	
Double 64-bit Byte Order	Order76543210	
Long 64-bit Byte Order	Order76543210	
Use Preset Multiple Register	false	
Use Force Multiple Coil	false	
Max Fails Until Device Down	2 [0 - max]	
Socket Option Timeout	0 h 1 m 0 s [1 ms - +inf]	
ModbusTcpDevice	ModbusTcpDevice[1]	

https://localhost/ord/station:%7Cslot:/Drivers/ModbusTningPoies

The ModbusTcpNetwork has the standard collection of network components, such as for status, health, monitor, tuning policies, and poll scheduler.

Other ModbusTcpNetwork properties such as Retry Count, Response Timeout, and Max Fails Until Device Down are typically left at defaults, unless particular reasons dictate the change.

You access local device properties by expanding **Drivers** > **ModbusTcpNetwork** in the Nav tree and double-clicking the ModbusTcpNetwork.

Property	Value	Description
Float Byte Order	drop-down list	Specifies the byte-order used to assemble or receive floating-point (32-bit) values in messages. Choices reflect two alternate methods, where numerals 0, 1, 2, and 3 represent the least significant byte to most significant byte: • `Order3210` – Most

Property	Value	Description
		significant byte first, or big-endian, it is the default. • `Order1032` – Bytes transmitted in a 1,0,3,2 order, or little–endian. • `Order0123` – Bytes transmitted in a 0,1,2,3 order, or little-endian.
Long Byte Order	drop-down list	Specifies the byte-order used to assemble or receive long integer (32-bit) values in messages. Choices reflect two alternate methods, where numerals 0, 1, 2, and 3 represent the least significant byte to most significant byte: • `Order3210` - Most significant byte first, or big-endian, it is the default. • `Order1032` - Bytes transmitted in a 1,0,3,2 order, or little-endian. • `Order0123` - Bytes transmitted in a 0,1,2,3 order, or little-endian. **NOTE:** Float or long values received in incorrect byte order may appear abnormally big, or not at all.
Double 64-bit Byte Order	drop-down list	Specifies the byte-order used to assemble or receive receives double-precision floating point (64-bit) values in messages. Choices reflect in 8 corresponding byte order options, where numerals 0, 1, 2, and 3 represent the least significant byte to most significant byte. When selecting the byte order options, it is required that the Float Byte Order and Long Byte Order are both set to `Order0123` to implement this configuration effectively: • `Order76543210` – Most significant byte first, or big-

Property	Value	Description
		endian order where the most significant byte is transmitted first (from 7 down to 0). • `Order 67452301` – Bytes transmitted in a order 6,7,4,5, etc or big-endian format involves swapping the bytes. • `Order54761032` – Bytes transmitted in a order 5, 4, 7, 6, etc or big-endian format but with each pair of bytes swapped (e.g., switching the first two bytes and the next two bytes). • `Order45670123` – Bytes are transmitted in order 4, 5, 6, 7, etc or big-endian with both words and bytes swapped. • `Order01234567` – Bytes are transmitted in order 0, 1, 2, 3, etc or little-endian (LE) arrangement where the bytes are transmitted in direct order from least significant to most significant (from 0 to 7). • `Order10325476` – Bytes are transmitted in order 1, 0, 3, 2, etc or little-endian format where the bytes are swapped during transmission. • `Order23016745` – Bytes are transmitted in order 2, 3, 0, 1, etc or little-endian format but with each pair of bytes swapped (e.g., switching the first two bytes and the next two bytes). • `Order32107654` – Bytes are transmitted in order 3, 2, 1, 0, etc or little-endian format, with both words and bytes swapped.
Long 64-bit Byte Order	drop-down list	Specifies the byte-order used to assemble or receive long integer (64-bit) values in messages.

Property	Value	Description
		Choices reflect in 8 corresponding byte order options, where numerals 0, 1, 2, and 3 represent the least significant byte to most significant byte. When selecting the byte order options, it is required that the Float Byte Order and Long Byte Order are both set to `Order0123` to implement this configuration effectively: • `Order76543210` – Most significant byte first, or big-endian (BE) order where the most significant byte is transmitted first (from 7 down to 0). • `Order 67452301` – Bytes transmitted in a order 6,7,4,5, etc or big-endian format involves swapping the bytes. • `Order54761032` – Bytes transmitted in a order 5, 4, 7, 6, etc or big-endian format but with each pair of bytes swapped (e.g., switching the first two bytes and the next two bytes). • `Order45670123` – Bytes are transmitted in order 4, 5, 6, 7, etc. or big-endian with both words and bytes swapped. • `Order01234567` – Bytes are transmitted in order 0, 1, 2, 3, etc or little-endian (LE) arrangement where the bytes are transmitted in direct order from least significant to most significant (from 0 to 7). • `Order10325476` – Bytes are transmitted in order 1, 0, 3, 2, etc or little-endian format where the bytes are swapped during transmission. • `Order23016745` – Bytes are transmitted in order 2, 3, 0, 1, etc or little-endian

Property	Value	Description
		format but with each pair of bytes swapped (e.g., switching the first two bytes and the next two bytes). • `Order32107654` – Bytes are transmitted in order 3, 2, 1, 0, etc or little-endian format, with both words and bytes swapped.
Use Force Multiple Coil	`true` or `false` (default)	Specifies whether to use function code 15 (Force Multiple Coils) instead of function code 05 (Force Single Coil) when writing to coils. The default is `false`, where function code 05 (Force Single Coil) is used. This property depends on the Modbus function codes supported by child devices, which (if available) provide alternative options in Modbus messaging. Function code 15 support (Preset Multiple Coils) is available in devices (true or false). The default is `false`, where function code **`Preset Single Coil`** is in place.
Use Preset Multiple Register	`true` or `false` (default)	Specifies whether to use function code 16 (**`Preset Multiple Registers`**) instead of function code 06 (**`Preset Single Register`**) when writing to registers. The default is `false`, where function code 06 (**`Preset Single Register`**) is used. This property depends on the Modbus function codes supported by child devices, which (if available) provide alternative options in Modbus messaging. Function code 16 support (**`Preset Multiple Registers`**) is available in devices (`true` or `false`). The default is `false`, where function code **`Preset Single Register`** is in place.

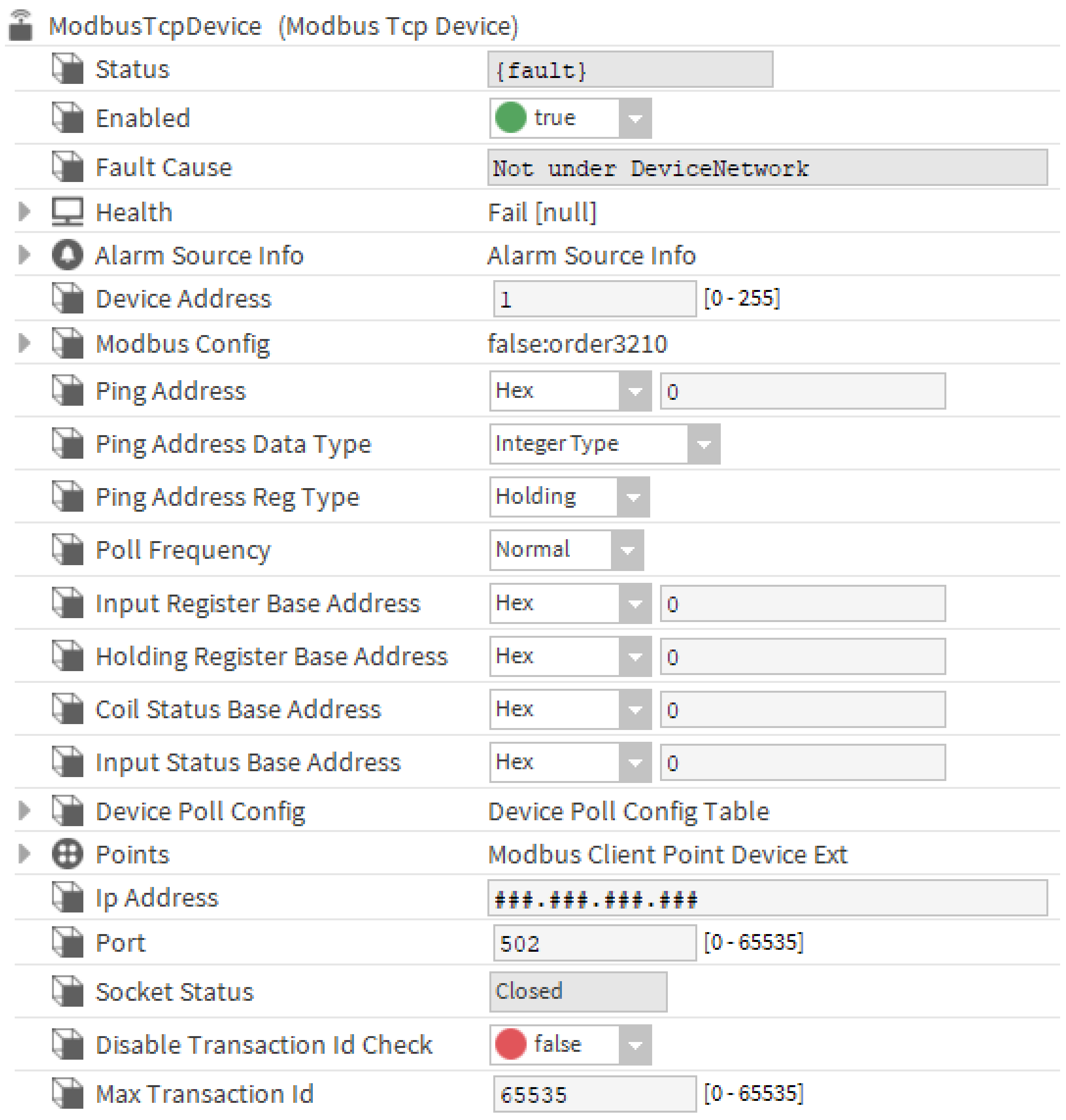
ModbusTcpDeviceFolder

This is the ModbusTcp implementation of a folder under a ModbusTcpNetwork. You can use these folders to organize ModbusTcpDevices in the network.
Typically, you add such folders using the **New Folder** button in the **Modbus Tcp Device Manager** view of the network. Each device folder has its own device manager view. The ModbusTcpDeviceFolder is also available in the **modbusTcp** palette.

ModbusTcpDevice

This component represents a Modbus TCP device under a ModbusTcpNetwork, for client access by the station (acting as Modbus master).

Figure 26. ModbusTcpDevice property sheet



ModbusTcpDevice (Modbus Tcp Device)

Status	{fault}
Enabled	● true
Fault Cause	Not under DeviceNetwork
Health	Fail [null]
Alarm Source Info	Alarm Source Info
Device Address	1 [0 - 255]
Modbus Config	false:order3210
Ping Address	Hex 0
Ping Address Data Type	Integer Type
Ping Address Reg Type	Holding
Poll Frequency	Normal
Input Register Base Address	Hex 0
Holding Register Base Address	Hex 0
Coil Status Base Address	Hex 0
Input Status Base Address	Hex 0
Device Poll Config	Device Poll Config Table
Points	Modbus Client Point Device Ext
Ip Address	###.###.###.###
Port	502 [0 - 65535]
Socket Status	Closed
Disable Transaction Id Check	● false
Max Transaction Id	65535 [0 - 65535]

Each client Modbus device object (ModbusAsyncDevice, ModbusTCPDevice, and ModbusTCPGatewayDevice) has an associated Modbus Config container slot to override these network-wide defaults. These properties adjust the settings for message transactions to (and from) only that device.

You access these properties by expanding **Drivers** > **ModbusTcpNetwork** and double-clicking the **ModbusTcpDevice** in the Nav tree.

In addition to the standard properties (Status Enabled, Fault Cause, Health and Alarm Source Info), these properties support the Modbus TCP device:

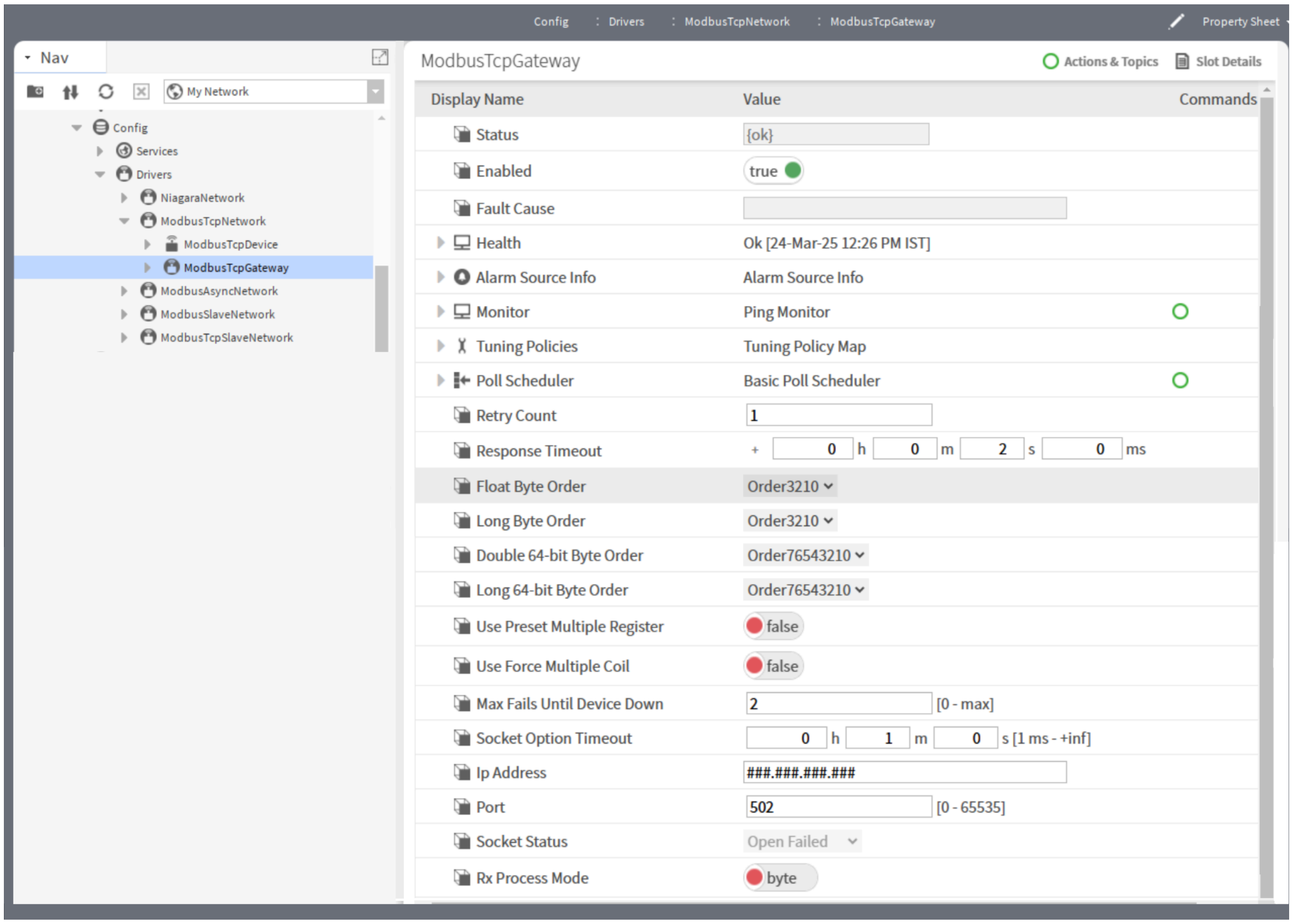
Property	Value	Description
Device Address	number from 1 to 247	Defines the unique number that identifies the current device object on the network.
Modbus Config	additional properties	Refer to Modbus Config .
Ping Address, Address Format	Hex (default), Decimal, or Modbus	Selects the format of an address used to automatically configure unique register addresses for the device's points. The driver merges this address with the point address.
Ping Address, Address	number	Defines the base address to use.
Ping Address Data Type	drop-down list	Defines the type of numeric data.
Ping Address Reg Type	drop-down list	Defines the type of register.
Poll Frequency	drop-down list	Configures how frequently the system polls proxy points.
Input Register Base Address, Address Format	Hex (default), Decimal, or Modbus	Selects the format of an address used to automatically configure unique register addresses for the device's points. The driver merges this address with the point address.
Input Register Base Address, Address	number	Defines the base address to use.
Holding Register Base Address, Address Format	Hex (default), Decimal, or Modbus	Selects the format of an address used to automatically configure unique register addresses for the device's points. The driver merges this address with the point address.
Holding Register Base Address, Address	number	Defines the base address to use.
Coil Status Base Address, Address Format	Hex (default), Decimal, or Modbus	Selects the format of an address used to automatically configure unique register addresses for the device's points. The driver merges this address with the point address.
Coil Status Base Address, Address	text	Defines the base address to use.
Input Status Base Address Format	Hex (default), Decimal, or Modbus	Selects the format of an address

Property	Value	Description
		used to automatically configure unique register addresses for the device's points. The driver merges this address with the point address.
Input Status Base Address	number	Defines the base address to use.
Device Poll Config	additional properties	Refer to Device Poll Config .
Points	additional properties	Provides a container for proxy points.
IP Address	IP address	Identifies a device, which is connected to a network that uses the Internet Protocol for communication.
Port	number (defaults to 502)	Specifies the TCP port used by Modbus message transactions. Leave at the default (502) unless the TCP/Ethernet-side of the Modbus TCP/serial gateway uses another TCP port.
Socket Status	read-only	Reports the current condition of the TCP/IP socket.
Disable Transaction Id Check	`true` or `false` (default)	Disables (`true`) and enables (`false`) use of the Transaction Id when synchronizing client/server messages. The most important bytes in a TCP frame are the Unit Identifier, Function Code and Data bytes. If your network does not need Transaction ID, use this property to disable it.
Max Transaction Id	number	Defines the largest number to assign as the Transaction ID.

ModbusTcpGateway

ModbusTcpGateway is the base container for one or more ModbusTcpGatewayDevice components. This network-level component specifies the TCP/IP address and port used to connect to the Modbus TCP/serial gateway, which has Modbus serial devices (typically Modbus RTU, via RS-485) on its far side. Those devices are represented by its child ModbusTcpGatewayDevices

Figure 27. ModbusTcpGateway property sheet



Display Name	Value	Commands
Status	{ok}	
Enabled	true	
Fault Cause		
Health	Ok [24-Mar-25 12:26 PM IST]	
Alarm Source Info	Alarm Source Info	
Monitor	Ping Monitor	
Tuning Policies	Tuning Policy Map	
Poll Scheduler	Basic Poll Scheduler	
Retry Count	1	
Response Timeout	+ 0 h 0 m 2 s 0 ms	
Float Byte Order	Order3210	
Long Byte Order	Order3210	
Double 64-bit Byte Order	Order76543210	
Long 64-bit Byte Order	Order76543210	
Use Preset Multiple Register	false	
Use Force Multiple Coil	false	
Max Fails Until Device Down	2 [0 - max]	
Socket Option Timeout	0 h 1 m 0 s [1 ms - +inf]	
Ip Address	###.###.###.###	
Port	502 [0 - 65535]	
Socket Status	Open Failed	
Rx Process Mode	byte	

The ModbusTcpGateway has the standard collection of network components, such as for status, health, monitor, tuning policies, and poll scheduler. Other ModbusTcpGateway properties such as Retry Count, Response Timeout, and Max Fails Until Device Down are typically left at defaults, unless particular reasons dictate the change.

You access these properties by expanding **Drivers** > **ModbusTcpNetwork** > **ModbusTcpDevice** and double-clicking the**ModbusTcpGateway** container in the Nav tree.

Property	Value	Description
Float Byte Order	drop-down list	Specifies the byte-order used to assemble or receive floating-point (32-bit) values in messages. Choices reflect two alternate methods, where numerals 0, 1, 2, and 3 represent the least significant byte to most significant byte: • `Order3210` – Most

Property	Value	Description
		significant byte first, or big-endian, it is the default. • `Order1032` – Bytes transmitted in a 1,0,3,2 order, or little–endian. • `Order0123` – Bytes transmitted in a 0,1,2,3 order, or little-endian.
Long Byte Order	drop-down list	Specifies the byte-order used to assemble or receive long integer (32-bit) values in messages. Choices reflect two alternate methods, where numerals 0, 1, 2, and 3 represent the least significant byte to most significant byte: • `Order3210` - Most significant byte first, or big-endian, it is the default. • `Order1032` - Bytes transmitted in a 1,0,3,2 order, or little-endian. • `Order0123` - Bytes transmitted in a 0,1,2,3 order, or little-endian. **NOTE:** Float or long values received in incorrect byte order may appear abnormally big, or not at all.
Double 64-bit Byte Order	drop-down list	Specifies the byte-order used to assemble or receive receives double-precision floating point (64-bit) values in messages. Choices reflect in 8 corresponding byte order options, where numerals 0, 1, 2, and 3 represent the least significant byte to most significant byte. When selecting the byte order options, it is required that the Float Byte Order and Long Byte Order are both set to `Order0123` to implement this configuration effectively: • `Order76543210` – Most significant byte first, or big-

Property	Value	Description
		endian order where the most significant byte is transmitted first (from 7 down to 0). • `Order 67452301` – Bytes transmitted in a order 6,7,4,5, etc or big-endian format involves swapping the bytes. • `Order54761032` – Bytes transmitted in a order 5, 4, 7, 6, etc or big-endian format but with each pair of bytes swapped (e.g., switching the first two bytes and the next two bytes). • `Order45670123` – Bytes are transmitted in order 4, 5, 6, 7, etc or big-endian with both words and bytes swapped. • `Order01234567` – Bytes are transmitted in order 0, 1, 2, 3, etc or little-endian (LE) arrangement where the bytes are transmitted in direct order from least significant to most significant (from 0 to 7). • `Order10325476` – Bytes are transmitted in order 1, 0, 3, 2, etc or little-endian format where the bytes are swapped during transmission. • `Order23016745` – Bytes are transmitted in order 2, 3, 0, 1, etc or little-endian format but with each pair of bytes swapped (e.g., switching the first two bytes and the next two bytes). • `Order32107654` – Bytes are transmitted in order 3, 2, 1, 0, etc or little-endian format, with both words and bytes swapped.
Long 64-bit Byte Order	drop-down list	Specifies the byte-order used to assemble or receive long integer (64-bit) values in messages.

Property	Value	Description
		Choices reflect in 8 corresponding byte order options, where numerals 0, 1, 2, and 3 represent the least significant byte to most significant byte. When selecting the byte order options, it is required that the Float Byte Order and Long Byte Order are both set to `Order0123` to implement this configuration effectively: • `Order76543210` – Most significant byte first, or big-endian (BE) order where the most significant byte is transmitted first (from 7 down to 0). • `Order 67452301` – Bytes transmitted in a order 6,7,4,5, etc or big-endian format involves swapping the bytes. • `Order54761032` – Bytes transmitted in a order 5, 4, 7, 6, etc or big-endian format but with each pair of bytes swapped (e.g., switching the first two bytes and the next two bytes). • `Order45670123` – Bytes are transmitted in order 4, 5, 6, 7, etc. or big-endian with both words and bytes swapped. • `Order01234567` – Bytes are transmitted in order 0, 1, 2, 3, etc or little-endian (LE) arrangement where the bytes are transmitted in direct order from least significant to most significant (from 0 to 7). • `Order10325476` – Bytes are transmitted in order 1, 0, 3, 2, etc or little-endian format where the bytes are swapped during transmission. • `Order23016745` – Bytes are transmitted in order 2, 3, 0, 1, etc or little-endian

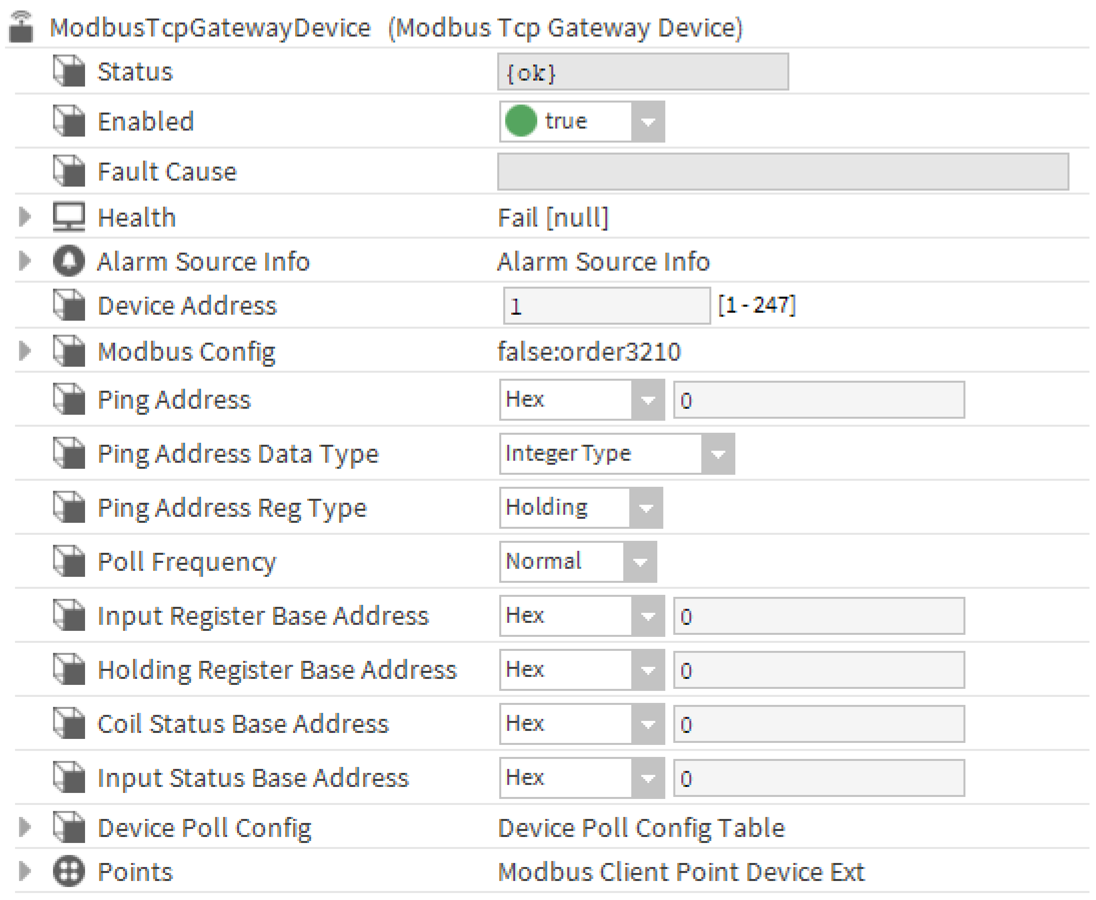
Property	Value	Description
		format but with each pair of bytes swapped (e.g., switching the first two bytes and the next two bytes). • `Order32107654` – Bytes are transmitted in order 3, 2, 1, 0, etc or little-endian format, with both words and bytes swapped.
Use Force Multiple Coil	`true` or `false` (default)	Specifies whether to use function code 15 (Force Multiple Coils) instead of function code 05 (Force Single Coil) when writing to coils. The default is `false`, where function code 05 (Force Single Coil) is used. This property depends on the Modbus function codes supported by child devices, which (if available) provide alternative options in Modbus messaging. Function code 15 support (Preset Multiple Coils) is available in devices (true or false). The default is `false`, where function code **`Preset Single Coil`** is in place.
Use Preset Multiple Register	`true` or `false` (default)	Specifies whether to use function code 16 (**`Preset Multiple Registers`**) instead of function code 06 (**`Preset Single Register`**) when writing to registers. The default is `false`, where function code 06 (**`Preset Single Register`**) is used. This property depends on the Modbus function codes supported by child devices, which (if available) provide alternative options in Modbus messaging. Function code 16 support (**`Preset Multiple Registers`**) is available in devices (`true` or `false`). The default is `false`, where function code **`Preset Single Register`** is in place.

Property	Value	Description
Ip Address	IP address	Identifies a device, which is connected to a network that uses the Internet Protocol for communication.
Port	number (defaults to 502)	Specifies the TCP port used by Modbus message transactions. Leave at the default (502) unless the TCP/Ethernet-side of the Modbus TCP/serial gateway uses another TCP port.

ModbusTcpGatewayDevice

The ModbusTcpGatewayDevice represents a Modbus serial (RTU or ASCII) device on the far side of a ModbusTcpGateway (network), for TCP client access by the station (acting as Modbus master).

Figure 28. ModbusTcpGatewayDevice property sheet


ModbusTcpGatewayDevice (Modbus Tcp Gateway Device)

Status	{ok}
Enabled	true
Fault Cause	
Health	Fail [null]
Alarm Source Info	Alarm Source Info
Device Address	1 [1 - 247]
Modbus Config	false:order3210
Ping Address	Hex 0
Ping Address Data Type	Integer Type
Ping Address Reg Type	Holding
Poll Frequency	Normal
Input Register Base Address	Hex 0
Holding Register Base Address	Hex 0
Coil Status Base Address	Hex 0
Input Status Base Address	Hex 0
Device Poll Config	Device Poll Config Table
Points	Modbus Client Point Device Ext

You can access these properties by expanding **Driver** > **ModbusTcpNetwork** and double-clicking the **ModbusTcpGatewayDevice**.

Property	Value	Description
Status	read-only	Reports the condition of the entity or process at last polling. `{ok}` indicates that the component is licensed and polling successfully. `{down}` indicates that the last check was unsuccessful, perhaps because of an incorrect property, or possibly loss of network connection. `{disabled}` indicates that the **`Enable`** property is set to `false`. `{fault}` indicates another problem. Refer to **`Fault Cause`** for more information.
Enabled	`true` or `false` (defaults to `true`)	Activates (`true`) and deactivates (`false`) use of the object (network, device, point, component, table, schedule, descriptor, etc.).
Fault Cause	read-only	Indicates the reason why a system object (network, device, component, extension, etc.) is not working (in fault). This property is empty unless a fault exists.
Health	read-only	Reports the status of the network, device or component. This advisory information, including a time stamp, can help you recognize and troubleshoot problems but it provides no direct management controls. The *Niagara Drivers Guide* documents the these properties.
Alarm Source Info	additional properties	Contains a set of properties for configuring and routing alarms when this component is the alarm source. For property descriptions, refer to the *Niagara Alarms Guide*

Property	Value	Description
Device Address	number from 1 to 247	Defines the unique number that identifies the current device object on the network.
Modbus Config	additional properties	Refer to Modbus Config .
Ping Address, Address Format	Hex (default), Decimal, or Modbus	Selects the format of an address used to automatically configure unique register addresses for the device's points. The driver merges this address with the point address.
Ping Address, Address	number	Defines the base address to use.
Ping Address Data Type	drop-down list	Defines the type of numeric data.
Ping Address Reg Type	drop-down list	Defines the type of register.
Poll Frequency	drop-down list	Configures how frequently the system polls proxy points.
Input Register Base Address, Address Format	Hex (default), Decimal, or Modbus	Selects the format of an address used to automatically configure unique register addresses for the device's points. The driver merges this address with the point address.
Input Register Base Address, Address	number	Defines the base address to use.
Holding Register Base Address, Address Format	Hex (default), Decimal, or Modbus	Selects the format of an address used to automatically configure unique register addresses for the device's points. The driver merges this address with the point address.
Holding Register Base Address, Address	number	Defines the base address to use.
Coil Status Base Address, Address Format	Hex (default), Decimal, or Modbus	Selects the format of an address used to automatically configure unique register addresses for the device's points. The driver merges this address with the point address.
Coil Status Base Address, Address	text	Defines the base address to use.
Input Status Base Address Format	Hex (default), Decimal, or Modbus	Selects the format of an address used to automatically configure unique register addresses for the device's points. The driver merges this address with the point address.
Input Status Base Address	number	Defines the base address to use.

Property	Value	Description
Device Poll Config	additional properties	Refer to Device Poll Config .
Points	additional properties	Refer to Points (client device) .

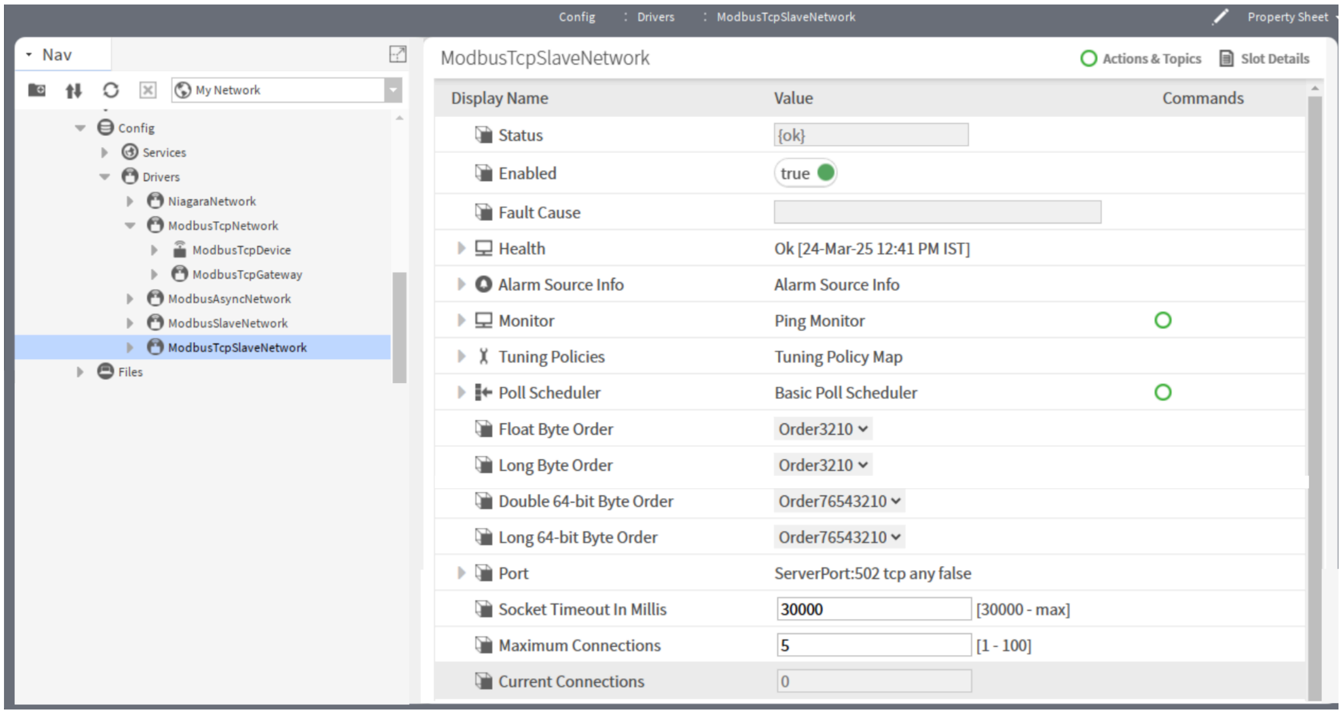
ModbusTcpGatewayDeviceFolder

This is the ModbusTcp implementation of a folder under a ModbusTcpGateway network. You can use these folders to organize ModbusTcpGatewayDevices in the network.
Typically, you add such folders using the **New Folder** button in the **ModbusTcp Gateway Device Manager** view of the network. Each device folder has its own device manager view. The ModbusTcpGatewayDeviceFolder is also available in the **modbusTcp** palette.

ModbusTcpSlaveNetwork

This component is the base container for all Tcp Slave components (devices and proxy points). It resides under the station **Driver** container. Its default view is the **Modbus Tcp Slave Device Manager**. In the property sheet of the ModbusTcpSlaveNetwork, you review the default global values for Modbus device data, and specify other TCP connection settings. You can specify many ranges of Modbus data items (coils, inputs, input registers, holding registers) in any or all ModbusTcpSlaveDevice components.

Figure 29. ModbusTcpSlaveNetwork property sheet



ModbusTcpSlaveNetwork

Display Name	Value	Commands
Status	{ok}	
Enabled	true ●	
Fault Cause		
Health	Ok [24-Mar-25 12:41 PM IST]	
Alarm Source Info	Alarm Source Info	
Monitor	Ping Monitor	○
Tuning Policies	Tuning Policy Map	
Poll Scheduler	Basic Poll Scheduler	○
Float Byte Order	Order3210	
Long Byte Order	Order3210	
Double 64-bit Byte Order	Order76543210	
Long 64-bit Byte Order	Order76543210	
Port	ServerPort:502 tcp any false	
Socket Timeout In Millis	30000 [30000 - max]	
Maximum Connections	5 [1 - 100]	
Current Connections	0	

You can access these properties by double-clicking **Driver** > **ModbusTcpSlaveNetwork.**

The ModbusTcpSlaveNetwork has the standard collection of network components, such as for status, health, monitor, tuning policies, and poll scheduler.

In addition, the following properties have special importance.

Property	Value	Description
Float Byte Order	drop-down list	Specifies the byte-order used to assemble or receive floating-point (32-bit) values in messages. Choices reflect two alternate methods, where numerals 0, 1, 2, and 3 represent the least significant byte to most significant byte: • `Order3210` – Most significant byte first, or big-endian, it is the default. • `Order1032` – Bytes transmitted in a 1,0,3,2 order, or little–endian. • `Order0123` – Bytes transmitted in a 0,1,2,3 order, or little-endian.
Long Byte Order	drop-down list	Specifies the byte-order used to assemble or receive long integer (32-bit) values in messages. Choices reflect two alternate methods, where numerals 0, 1, 2, and 3 represent the least significant byte to most significant byte: • `Order3210` - Most significant byte first, or big-endian, it is the default. • `Order1032` - Bytes transmitted in a 1,0,3,2 order, or little-endian. • `Order0123` - Bytes transmitted in a 0,1,2,3 order, or little-endian. **NOTE:** Float or long values received in incorrect byte order may appear abnormally big, or not at all.
Double 64-bit Byte Order	drop-down list	Specifies the byte-order used to assemble or receive receives double-precision floating point (64-bit) values in messages. Choices reflect in 8 corresponding byte order options, where numerals 0, 1, 2, and 3 represent the least

Property	Value	Description
		significant byte to most significant byte. When selecting the byte order options, it is required that the Float Byte Order and Long Byte Order are both set to `Order0123` to implement this configuration effectively: • `Order76543210` – Most significant byte first, or big-endian order where the most significant byte is transmitted first (from 7 down to 0). • `Order 67452301` – Bytes transmitted in a order 6,7,4,5, etc or big-endian format involves swapping the bytes. • `Order54761032` – Bytes transmitted in a order 5, 4, 7, 6, etc or big-endian format but with each pair of bytes swapped (e.g., switching the first two bytes and the next two bytes). • `Order45670123` – Bytes are transmitted in order 4, 5, 6, 7, etc or big-endian with both words and bytes swapped. • `Order01234567` – Bytes are transmitted in order 0, 1, 2, 3, etc or little-endian (LE) arrangement where the bytes are transmitted in direct order from least significant to most significant (from 0 to 7). • `Order10325476` – Bytes are transmitted in order 1, 0, 3, 2, etc or little-endian format where the bytes are swapped during transmission. • `Order23016745` – Bytes are transmitted in order 2, 3, 0, 1, etc or little-endian format but with each pair of bytes swapped (e.g., switching the first two bytes and the next two bytes).

Property	Value	Description
		• `Order32107654` – Bytes are transmitted in order 3, 2, 1, 0, etc or little-endian format, with both words and bytes swapped.
Long 64-bit Byte Order	drop-down list	Specifies the byte-order used to assemble or receive long integer (64-bit) values in messages. Choices reflect in 8 corresponding byte order options, where numerals 0, 1, 2, and 3 represent the least significant byte to most significant byte. When selecting the byte order options, it is required that the Float Byte Order and Long Byte Order are both set to `Order0123` to implement this configuration effectively: • `Order76543210` – Most significant byte first, or big-endian (BE) order where the most significant byte is transmitted first (from 7 down to 0). • `Order 67452301` – Bytes transmitted in a order 6,7,4,5, etc or big-endian format involves swapping the bytes. • `Order54761032` – Bytes transmitted in a order 5, 4, 7, 6, etc or big-endian format but with each pair of bytes swapped (e.g., switching the first two bytes and the next two bytes). • `Order45670123` – Bytes are transmitted in order 4, 5, 6, 7, etc. or big-endian with both words and bytes swapped. • `Order01234567` – Bytes are transmitted in order 0, 1, 2, 3, etc or little-endian (LE) arrangement where the bytes are transmitted in direct order from least significant to most

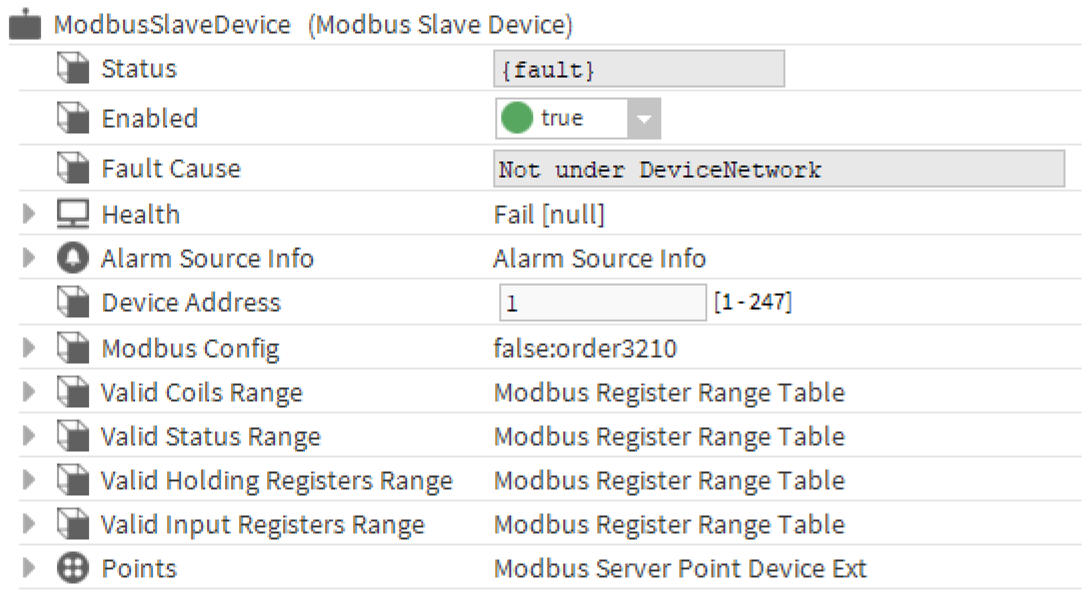
Property	Value	Description
		significant (from 0 to 7). • `Order10325476` – Bytes are transmitted in order 1, 0, 3, 2, etc or little-endian format where the bytes are swapped during transmission. • `Order23016745` – Bytes are transmitted in order 2, 3, 0, 1, etc or little-endian format but with each pair of bytes swapped (e.g., switching the first two bytes and the next two bytes). • `Order32107654` – Bytes are transmitted in order 3, 2, 1, 0, etc or little-endian format, with both words and bytes swapped.
Port	number (defaults to 502)	Specifies the TCP port used by Modbus message transactions. 502 is the standard Modbus TCP port. Leave at the default (502) unless the Modbus TCP master uses another TCP port.
Socket Timeout In Millis	defaults to 30000 (milliseconds, or 30 seconds—the minimum).	You can adjust upwards if necessary.
Maximum Connections	number	Specifies the maximum connection.

ModbusTcpSlaveDevice

The ModbusTcpSlaveDevice represents a virtual Modbus device to serve data to a Modbus master over a TCP connection, where station data appears as Modbus data items. It has 4 frozen range containers (ModbusRegisterRangeTables), which specify what Modbus addresses are available as coils, inputs, holding registers, and input registers.

Its Points extension (ModbusServerPointDeviceExt) contains Modbus proxy points with server proxy extensions (ModbusServerBooleanProxyExt, ModbusServerNumericProxyExt, ModbusServerRegisterBitProxyExt), used to read and write data to the defined data items. The device can also contain one or more ModbusServerStringRecord components.

Figure 30. ModbusTcpSlaveDevice property sheet


ModbusSlaveDevice (Modbus Slave Device)

Status	{fault}
Enabled	true
Fault Cause	Not under DeviceNetwork
Health	Fail [null]
Alarm Source Info	Alarm Source Info
Device Address	1 [1 - 247]
Modbus Config	false:order3210
Valid Coils Range	Modbus Register Range Table
Valid Status Range	Modbus Register Range Table
Valid Holding Registers Range	Modbus Register Range Table
Valid Input Registers Range	Modbus Register Range Table
Points	Modbus Server Point Device Ext

You can access these properties by expanding **Driver** > **ModbusSlaveNetwork** and double-clicking the **ModbusSlaveDevice**.

Property	Value	Description
Status	read-only	Reports the condition of the entity or process at last polling. `{ok}` indicates that the component is licensed and polling successfully. `{down}` indicates that the last check was unsuccessful, perhaps because of an incorrect property, or possibly loss of network connection. `{disabled}` indicates that the **`Enable`** property is set to `false`. `{fault}` indicates another problem. Refer to **`Fault Cause`** for more information.
Enabled	`true` or `false` (defaults to `true`)	Activates (`true`) and deactivates (`false`) use of the object (network, device, point, component, table, schedule, descriptor, etc.).

Property	Value	Description
Fault Cause	read-only	Indicates the reason why a system object (network, device, component, extension, etc.) is not working (in fault). This property is empty unless a fault exists.
Health	read-only	Reports the status of the network, device or component. This advisory information, including a time stamp, can help you recognize and troubleshoot problems but it provides no direct management controls. The *Niagara Drivers Guide* documents the these properties.
Alarm Source Info	additional properties	Contains a set of properties for configuring and routing alarms when this component is the alarm source. For property descriptions, refer to the *Niagara Alarms Guide*
Device Address	number from 1 to 247	Defines the unique number that identifies the current device object on the network.
Modbus Config	additional properties	Refer to Modbus Config .
Valid Coils Range	additional properties	Refer to ValidCoilsRange .
Valid Status Range	additional properties	Refer to ValidCoilsRange for properties.
Valid Holding Registers Range	additional properties	Refer to ValidCoilsRange for properties.
Valid Input Registers Range	additional properties	Refer to ValidCoilsRange for properties.
Points	additional properties	Provides a container for proxy points.

Chapter 8. Plugins (views)

Plugins provide views of components and can be accessed in many ways. For example, double-click a component in the Nav tree to see its default view. In addition, you can right-click on a component and select from its **Views** menu.

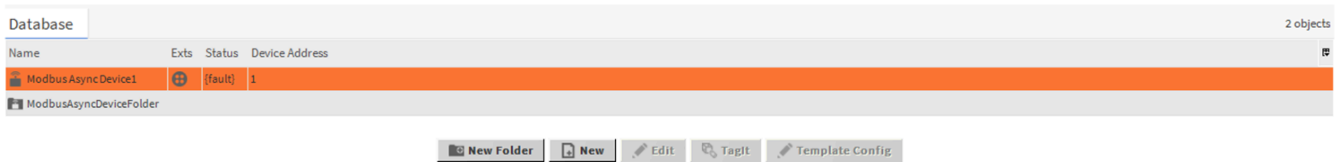
For summary documentation on any view, select **Help** > **On View** (**F1**) from the menu or press **F1** while the view is open.

Modbus Async Device Manager

This view creates and edits Modbus async devices.

Columns

Figure 31. Modbus Async Device Manager



Database — 2 objects

Name	Exts	Status	Device Address
ModbusAsyncDevice1		{fault}	1
ModbusAsyncDeviceFolder			

To view, double-click the **ModbusAsyncNetwork** node in the Nav tree or right-click this node and click **Views** > **Modbus Async Device Manager**.

Column	Description
Name	Displays the name of the Async device.
Type	Displays the type of the Async device.
Exts	Displays the point extension.
Status	Indicates the condition at the last check.
Enabled	Displays the Enabled status of the device.
Health	Displays the health of the device.
Device Address	Displays the device address.
Modbus Config	Displays the settings for message transactions to (and from) the device.
Poll Frequency	Displays the poll services that is used in the device.
Ping Address	Displays the ping address of the device.
Ping address Data Type	Displays the data type.

Buttons

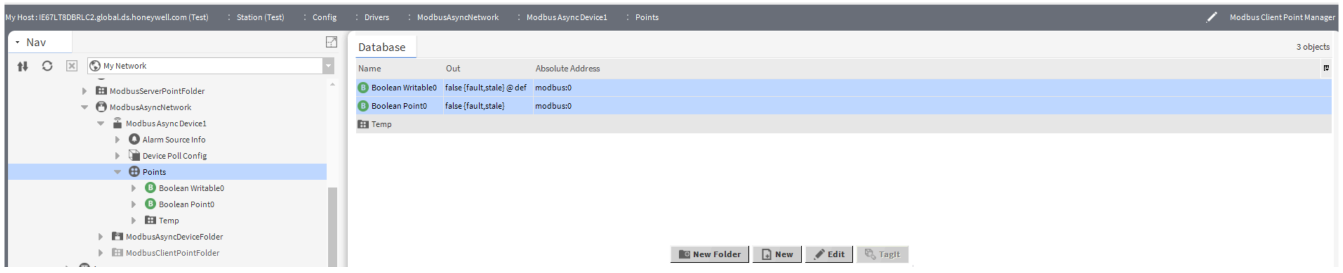
Button	Description
New Folder	Adds new folder in the Modbus Network
New	Adds a new device
Edit	Edits the added devices

Modbus Client Point Manager

This view creates and edits proxy points under a client Modbus device. It is the default view on the **Points** container of a **ModbusAsyncDevice**, **ModbusTcpDevice**, and **ModbusTcpGatewayDevice**, as well as on any points folder under these **Points** containers.

Columns

Figure 32. Modbus Client Point Manager view



Name	Out	Absolute Address
Boolean Writable0	false {fault,stale} @ def	modbus:0
Boolean Point0	false {fault,stale}	modbus:0
Temp		

To access this view, expand **Config** > **Drivers** > **ModbusTcpNetwork** or **ModbusAsyncNetwork** , expand **ModbusAsyncDevice** or **ModbusTcpDevice** and double-click **Points**

NOTE: Unlike the point managers in many other drivers, the **Modbus Client Point Manager** offers no learn mode. The protocol’s simplicity makes these functions unnecessary. Instead, you use the **New** button to create proxy points, referring to the vendor’s documentation for the addresses of data items in each Modbus device.

By default, only a few of the available columns in the **Modbus Client Point Manager are** enabled for display, notably Name, Out and Absolute Address. However, you may wish to change this by clicking on the Table Options menu in the table’s upper right. For example, during the configuration process you may wish to see Fault Cause and Data Source.

Column	Description
Path	Displays the ord path of point.
Name	Displays the name of point.
Type	Indicates the type of point.
Out	Reports the current out value, including any point facets. This defaults to the single (configured) property value along with status for the proxy point.
Enabled	Reports if the proxy point is currently enabled for communication.
Tuning Policy Name	Displays the name of the tuning policy
Absolute Address	Displays the actual address that uses when polling for this discrete data point from the actual Modbus device.
Fault Cause	Indicates the condition at the last check.
Poll Frequency	Displays the poll services that is used in the device.
Data Address	Displays the address of the polled data item
Reg Type	Displays the register type.
Data Type	Displays the location of data.
Status Type	Displays the location of points.
Bit Number	Displays the bit number.
Beginning Bit	Displays the bit in the register.
Number of Bits	Displays the number of bits used for point.
Data Source	Displays the data source.
Number Registers	Displays the number of consecutive holding registers to read.
Device Facets	Represents the facets learned from the point.
Facets	Displays the facet value.
Conversion	Displays the units to use when converting values from the device facets to point facets.
Read Value	Displays the value read by the driver.

Column	Description
Write Value	Displays the value to be written.

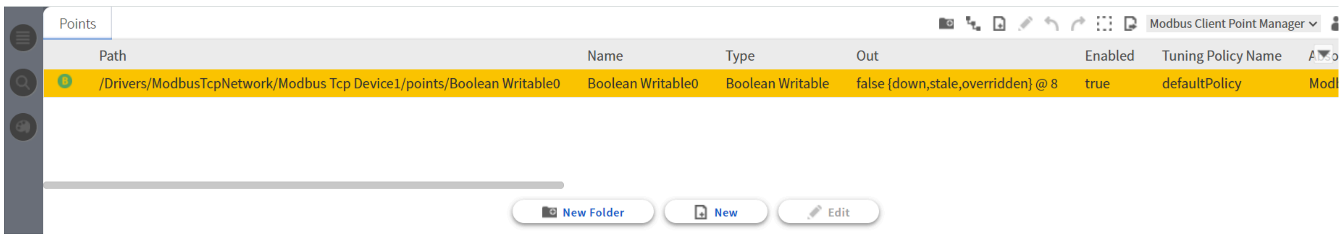
Buttons

- **New Folder** creates a new folder for devices. Each such folder provides its own set of manager views.
- **New** creates a new device record in the database.
- **Edit** opens the device's database record for updating.

HTML 5 Modbus Client Point Ux Manager

In Niagara 4.14 and later, there is added browser support for Modbus Tcp Client Device Manager View. The HTML 5 version of this view is a web-browser-based implementation and it provides the same functions as the Workbench view. This view creates and edits proxy points under a client Modbus device. It is the default view on the **Points** container of a **ModbusAsyncDevice**, **ModbusTcpDevice**, as well as on any points folder under these **Points** containers.

Figure 33. Modbus Client Point Ux Manager



Path	Name	Type	Out	Enabled	Tuning Policy Name	A
/Drivers/ModbusTcpNetwork/Modbus Tcp Device1/points/Boolean Writable0	Boolean Writable0	Boolean Writable	false {down,stale,overridden} @ 8	true	defaultPolicy	Mod

To access this view, expand **Config** > **Drivers** > **ModbusTcpNetwork** or **ModbusAsyncNetwork** , expand **ModbusAsyncDevice** or **ModbusTcpDevice** and click **Points**

Columns

Column	Description
Path	Displays the ord path of point.
Name	Displays the name of point.
Type	Indicates the type of point.
Out	Reports the current out value, including any point facets. This defaults to the single (configured) property value along with status for the proxy point.
Enabled	Reports if the proxy point is currently enabled for communication.
Tuning Policy Name	Displays the name of the tuning policy
Absolute Address	Displays the actual address that uses when polling for this discrete data point from the actual Modbus device.
Fault Cause	Indicates the condition at the last check.
Poll Frequency	Displays the poll services that is used in the device.
Data Address	Displays the address of the polled data item
Reg Type	Displays the register type.
Data Type	Displays the location of data.
Status Type	Displays the location of points.
Bit Number	Displays the bit number.
Beginning Bit	Displays the bit in the register.
Number of Bits	Displays the number of bits used for point.
Data Source	Displays the data source.

Column	Description
Number Registers	Displays the number of consecutive holding registers to read.
Device Facets	Represents the facets learned from the point.
Facets	Displays the facet value.
Conversion	Displays the units to use when converting values from the device facets to point facets.
Read Value	Displays the value read by the driver.
Write Value	Displays the value to be written.

Buttons

- **New Folder** creates a new folder for devices. Each such folder provides its own set of manager views.
- **New** creates a new device record in the database.
- **Edit** opens the device's database record for updating.

Toolbar for Modbus Tcp Client Point Ux Manager

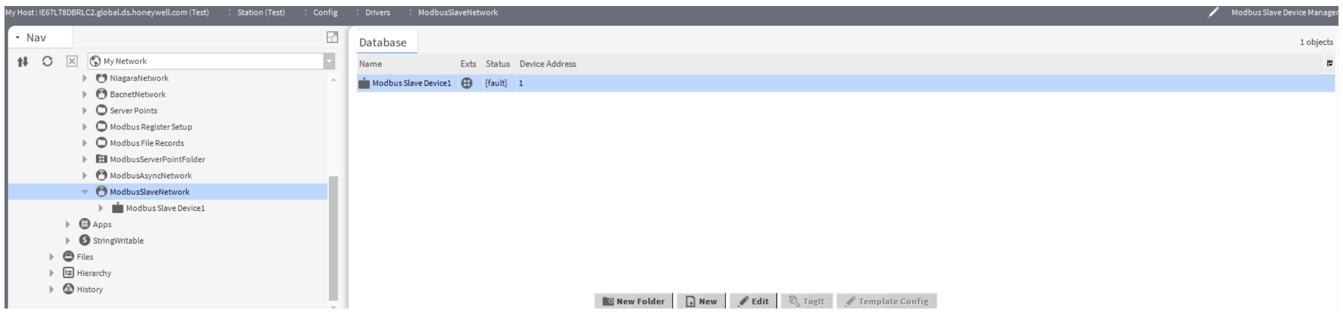
Toolbar Icon and Name	Description
New Folder	It creates a new folder in the database pane.
Trace	Displays all descendants or immediate children of the selected parent proxy point.
Create new objects	It creates new object in the database pane.
Edit objects	Opens the device's database record for updating.
Undo	Reverses the previous command.
Redo	Restores a command action after the Undo command has removed it.
Multi-selection Mode	Enables you to individually select multiple points without holding down the ctrl key.
Export	Exports the current view or object.

Modbus Slave Device Manager

This view creates and edits Modbus Slave Devices. It is the default view for the **ModbusSlaveNetwork**.

Columns

Figure 34. Modbus Slave Device Manager view



Name	Exts	Status	Device Address
Modbus Slave Device1		{fault}	1

To access this view, double-click a **ModbusSlaveNetwork** node in the Nav tree or right-click and select **Views** > **Modbus Slave Device Manager**.

Column	Description
Name	Displays the name of the slave device.
Type	Displays the type of the slave device.
Exts	Displays the point extension.
Status	Indicates the condition at the last check.
Enabled	Displays the Enabled status of the device.
Health	Displays the health of the device.
Device Address	Displays the device address.
Modbus Config	Displays the settings for message transactions to (and from) the device.
Poll Frequency	Displays the poll services that is used in the device.
Ping Address	Displays the ping address of the device.
Ping address Data Type	Displays the data type.

Buttons

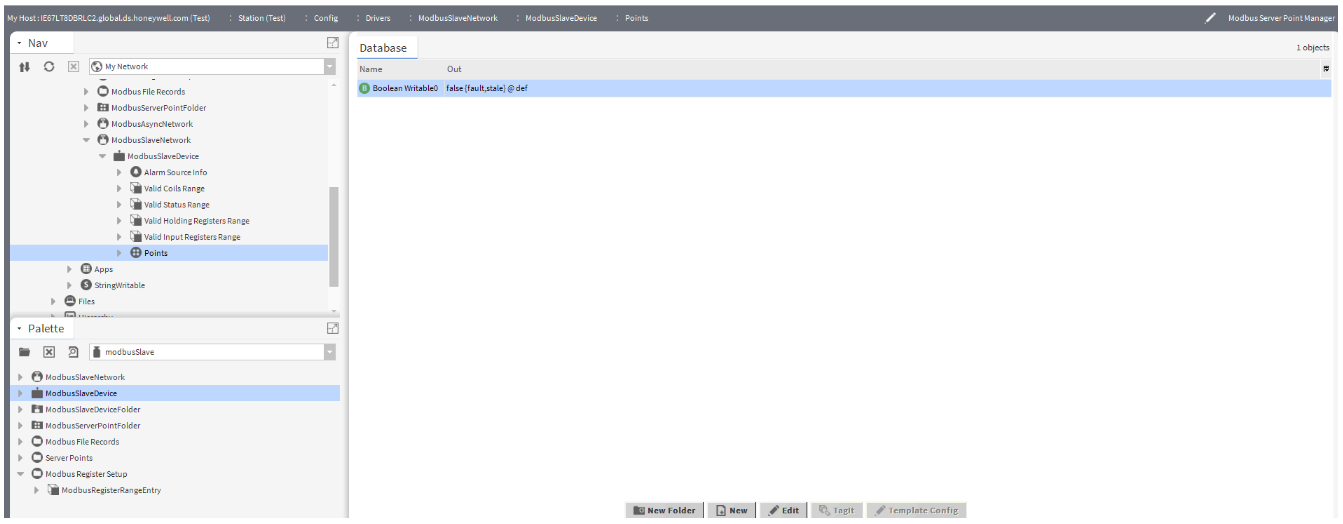
Button	Description
New Folder	Adds new folder in the Modbus Network
New	Adds a new device
Edit	Edits the added devices

Modbus Server Point Manager

This view creates and edits proxy points under a server Modbus device. It is the default view on the **Points** container for a **ModbusSlaveDevice** and **ModbusTcpSlaveDevice**, as well as on any points folder under those **Points** containers.

Buttons

Figure 35. Modbus Server Point Manager view



Name	Out
Boolean Writable0	false {fault,stale} @ def

To access this view, expand **Config** > **Drivers** > **ModbusSlaveNetwork** or **ModbusTcpSlaveNetwork** , expand **ModbusClientDevice** or **ModbusTcpClientDevice** and click **Points**

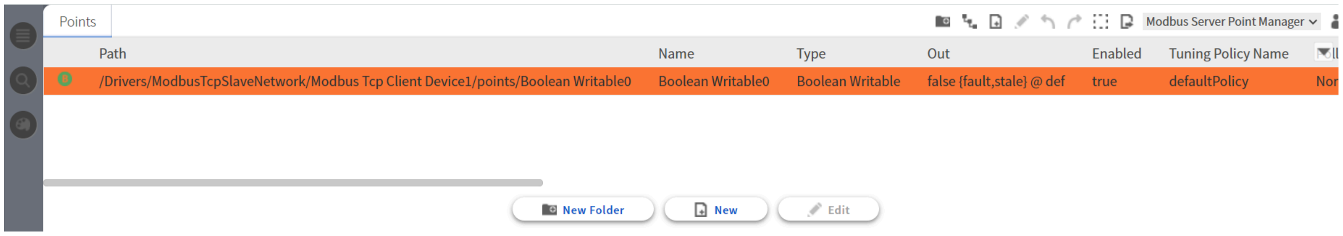
Column	Description
Path	Displays the ord path of point.
Name	Displays the name of point.
Type	Indicates the type of point.
Out	Reports the current out value, including any point facets. This defaults to the single (configured) property value along with status for the proxy point.
Enabled	Reports if the proxy point is currently enabled for communication.
Tuning Policy Name	Displays the name of the tuning policy
Absolute Address	Displays the actual address that uses when polling for this discrete data point from the actual Modbus device.
Fault Cause	Indicates the condition at the last check.
Poll Frequency	Displays the poll services that is used in the device.
Data Address	Displays the address of the polled data item
Reg Type	Displays the register type.
Data Type	Displays the location of data.
Status Type	Displays the location of points.
Bit Number	Displays the bit number.
Beginning Bit	Displays the bit in the register.
Number of Bits	Displays the number of bits used for point.
Data Source	Displays the data source.
Number Registers	Displays the number of consecutive holding registers to read.
Device Facets	Represents the facets learned from the point.
Facets	Displays the facet value.
Conversion	Displays the units to use when converting values from the device facets to point facets.
Read Value	Displays the value read by the driver.
Write Value	Displays the value to be written.

- **New Folder** creates a new folder for devices. Each such folder provides its own set of manager views.
- **New** creates a new device record in the database.
- **Edit** opens the device's database record for updating.

HTML 5 Modbus Server Point Ux Manager

In Niagara 4.14 and later, there is added browser support for Modbus Tcp Client Device Manager View. The HTML 5 version of this view is a web-browser-based implementation and it provides the same functions as the Workbench view. This view creates and edits proxy points under a server Modbus device. It is the default view on the **Points** container for a **ModbusClientDevice** and **ModbusTcpClientDevice**, as well as on any points folder under those **Points** containers.

Figure 36. Modbus Server Point Ux Manager



Path	Name	Type	Out	Enabled	Tuning Policy Name	
/Drivers/ModbusTcpSlaveNetwork/Modbus Tcp Client Device1/points/Boolean Writable0	Boolean Writable0	Boolean Writable	false {fault,stale} @ def	true	defaultPolicy	Nor

To access this view, expand **Config** > **Drivers** > **ModbusSlaveNetwork** or **ModbusTcpSlaveNetwork** , expand **ModbusClientDevice** or **ModbusTcpClientDevice** and click **Points**

Columns

Column	Description
Path	Displays the ord path of point.
Name	Displays the name of point.
Type	Indicates the type of point.
Out	Reports the current out value, including any point facets. This defaults to the single (configured) property value along with status for the proxy point.
Enabled	Reports if the proxy point is currently enabled for communication.
Tuning Policy Name	Displays the name of the tuning policy
Absolute Address	Displays the actual address that uses when polling for this discrete data point from the actual Modbus device.
Fault Cause	Indicates the condition at the last check.
Poll Frequency	Displays the poll services that is used in the device.
Data Address	Displays the address of the polled data item
Reg Type	Displays the register type.
Data Type	Displays the location of data.
Status Type	Displays the location of points.
Bit Number	Displays the bit number.
Beginning Bit	Displays the bit in the register.
Number of Bits	Displays the number of bits used for point.
Data Source	Displays the data source.
Number Registers	Displays the number of consecutive holding registers to read.
Device Facets	Represents the facets learned from the point.
Facets	Displays the facet value.
Conversion	Displays the units to use when converting values from the device facets to point facets.

Column	Description
Read Value	Displays the value read by the driver.
Write Value	Displays the value to be written.

Buttons

- **New Folder** creates a new folder for devices. Each such folder provides its own set of manager views.
- **New** creates a new device record in the database.
- **Edit** opens the device's database record for updating.

Toolbar for Server Point Ux Manager

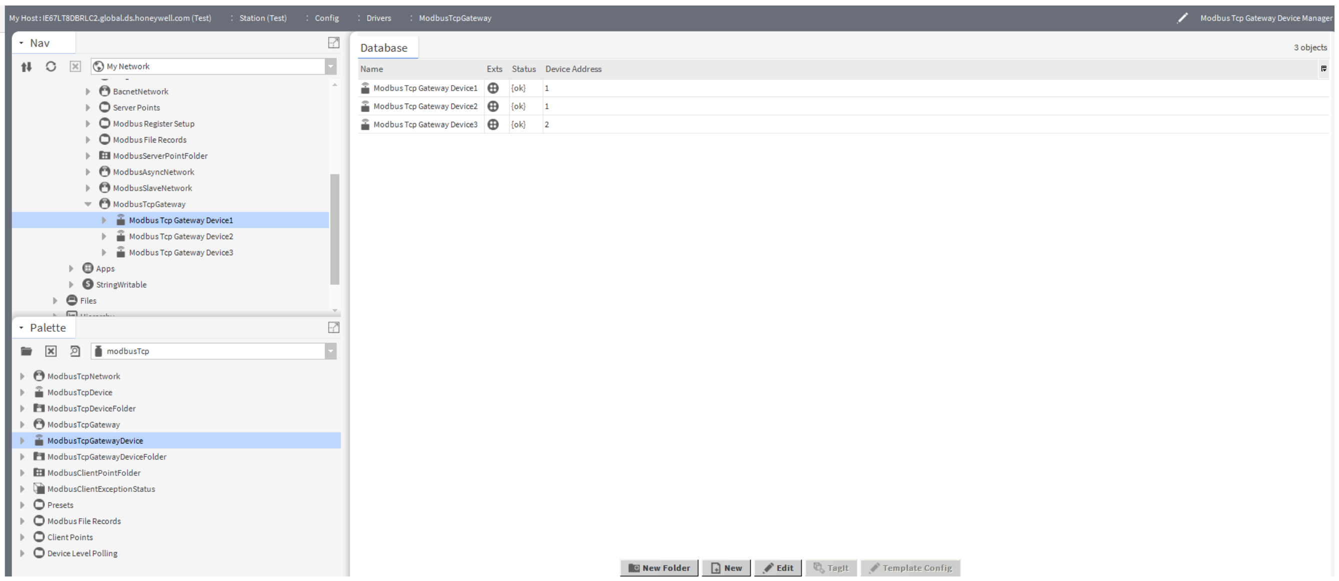
Toolbar Icon and Name	Description
New Folder	It creates a new folder in the database pane.
Trace	Displays all descendants or immediate children of the selected parent proxy point.
Create new objects	It creates new object in the database pane.
Edit objects	Opens the device's database record for updating.
Undo	Reverses the previous command.
Redo	Restores a command action after the Undo command has removed it.
Multi-selection Mode	Enables you to individually select multiple points without holding down the ctrl key.
Export	Exports the current view or object.

Modbus Tcp Gateway Device Manager

This view creates and edits Modbus Tcp Gateway Devices. It is the default view on the **ModbusTcpGateway**.

Columns

Figure 37. Modbus Tcp Gateway Device Manager view



Name	Exts	Status	Device Address
Modbus Tcp Gateway Device1		{ok}	1
Modbus Tcp Gateway Device2		{ok}	1
Modbus Tcp Gateway Device3		{ok}	2

To access this view, double-click a **ModbusTcpGateway** or right-click and select **Views** > **Modbus Tcp Gateway Device Manager**.

Column	Description
Name	Displays the name of the device.
Type	Displays the type of the device.
Exts	Displays the point extension.
Status	Indicates the condition at the last check.
Enabled	Displays the Enabled status of the device.
Health	Displays the health of the device.
Device Address	Displays the device address.
Modbus Config	Displays the settings for message transactions to (and from) the device.
Poll Frequency	Displays the poll services that is used in the device.
Ping Address	Displays the ping address of the device.
Ping address Data Type	Displays the data type.

Buttons

Button	Description
New Folder	Adds new folder in the Modbus Network
New	Adds a new device
Edit	Edits the added devices

Modbus Tcp Slave Device Manager

This view creates and edits Modbus Tcp Slave Devices. It is the default view on the **ModbusTcpSlaveNetwork**.

Columns

To access this view, double-click a **ModbusTcpSlaveNetwork** or right-click and select **Views** > **Modbus Tcp Slave Device Manager**.

Column	Description
Name	Displays the name of the device.
Type	Displays the type of the device.
Exts	Displays the point extension.
Status	Indicates the condition at the last check.
Enabled	Displays the Enabled status of the device.
Health	Displays the health of the device.
Device Address	Displays the device address.
Modbus Config	Displays the settings for message transactions to (and from) the device.
Poll Frequency	Displays the poll services that is used in the device.
Ping Address	Displays the ping address of the device.
Ping address Data Type	Displays the data type.

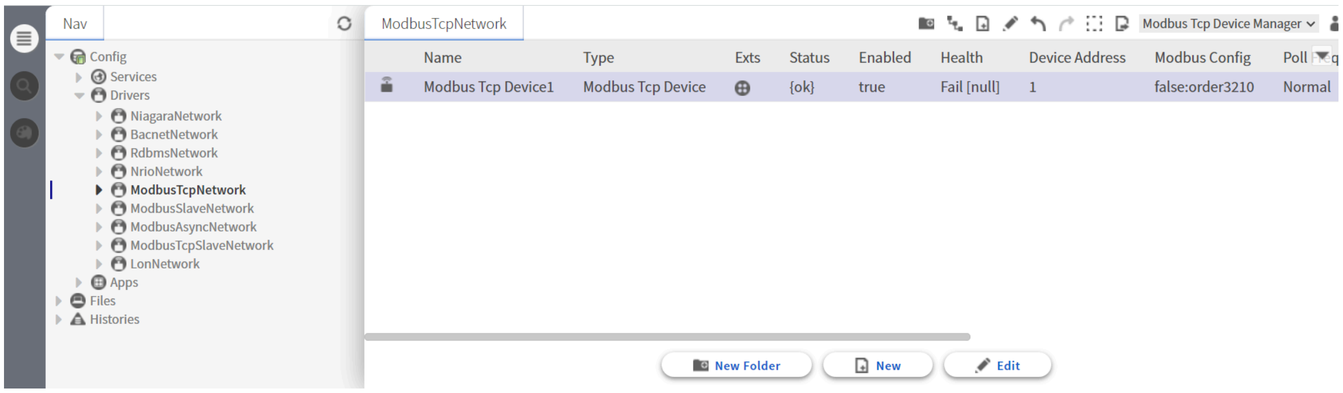
Buttons

Button	Description
New Folder	Adds new folder in the Modbus Network
New	Adds a new device
Edit	Edits the added devices

HTML5- Modbus Tcp Device Ux Manager

In Niagara 4.14 and later, there is added browser support for Modbus Tcp Device Manager View. The HTML 5 version of this view is a web-browser-based implementation and it provides the same functions as the Workbench view.

Figure 38. ModbusTcpUxDeviceUxManager



Name	Type	Exts	Status	Enabled	Health	Device Address	Modbus Config	Poll
Modbus Tcp Device1	Modbus Tcp Device		{ok}	true	Fail [null]	1	false:order3210	Normal

To access this view, expand **Config** > **Drivers** and double-click **ModbusTcpNetwork** or right-click **ModbusTcpNetwork** > **Views** > **Modbus Tcp Device Manager**.

Columns

Column Name	Description
Name	Reports the name of the entity or logical grouping.
Type	Displays the type of database.
Exts	Displays the device extension’s hyperlinks, including: Points, Alarms, Schedules, Trend Logs and Config.
Enabled	Indicates if the network, device, point or component is active or inactive.

Column Name	Description
Status	Reports the current condition of the entity as of the last refresh: {alarm}, {disabled}, {down}, {fault}, {ok}, {stale}, {unackedAlarm}
Health	Displays the status of the network, device or component.
Device Address	Displays the unique number that identifies the current device object on the network
Modbus Config	Displays the settings for message transactions to (and from) only that device.
Poll Frequency	Displays how frequently the system polls proxy points.
Ping Address	Displays the base address to use.
Ping Address Data Type	Displays the type of numeric data.
IP Address	Displays the Ip address of the device, which is connected to a network that uses the Internet Protocol for communication.
Port	Displays the TCP port used by Modbus message transaction.
Socket Status	Displays the current condition of the TCP/IP socket.

Buttons

- **New Folder** creates a new folder for devices. Each such folder provides its own set of manager views.
- **New** creates a new device record in the database.
- **Edit** opens the device's database record for updating.

Toolbar for Modbus Tcp Device Ux Manager

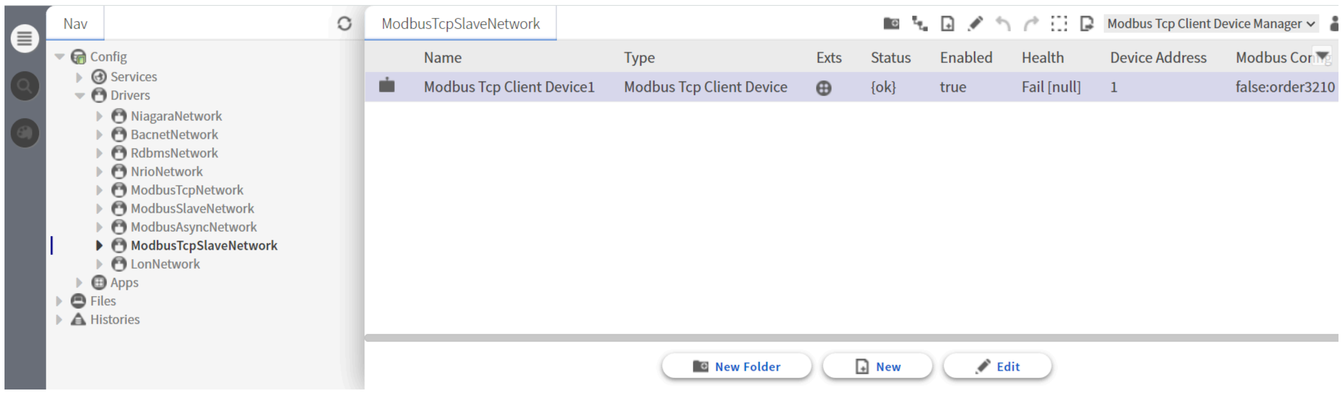
Toolbar Icon and Name	Description
New Folder	It creates a new folder in the database pane.
Trace	Displays all descendants or immediate children of the selected parent proxy point.
Create new objects	It creates new object in the database pane.
Edit objects	Opens the device's database record for updating.
Undo	Reverses the previous command.
Redo	Restores a command action after the Undo command has removed it.
Multi-selection Mode	Enables you to individually select multiple points without holding down the ctrl key.

Toolbar Icon and Name	Description
Export	Exports the current view or object.

HTML5- Modbus Tcp Client Device Ux Manager

In Niagara 4.14 and later, there is added browser support for Modbus Tcp Client Device Manager View. The HTML 5 version of this view is a web-browser-based implementation and it provides the same functions as the Workbench view.

Figure 39. ModbusTcpClientUxDeviceUxManager



Name	Type	Exts	Status	Enabled	Health	Device Address	Modbus Cor
Modbus Tcp Client Device1	Modbus Tcp Client Device		{ok}	true	Fail [null]	1	false:order3210

To access this view, expand **Config** > **Drivers** and double-click **ModbusTcpSlaveNetwork** or right-click **ModbusTcpSlaveNetwork** > **Views** > **Modbus Tcp Client Device Manager**.

Columns

Column Name	Description
Name	Reports the name of the entity or logical grouping.
Type	Displays the type of database.
Exts	Displays the device extension’s hyperlinks, including: Points, Alarms, Schedules, Trend Logs and Config.
Enabled	Indicates if the network, device, point or component is active or inactive.
Status	Reports the current condition of the entity as of the last refresh: {alarm}, {disabled}, {down}, {fault}, {ok}, {stale}, {unackedAlarm}
Health	Displays the status of the network, device or component.
Device Address	Displays the unique number that identifies the current device object on the network
Modbus Config	Displays the settings for message transactions to (and from) only that device.

Buttons

- **New Folder** creates a new folder for devices. Each such folder provides its own set of manager views.
- **New** creates a new device record in the database.
- **Edit** opens the device’s database record for updating.

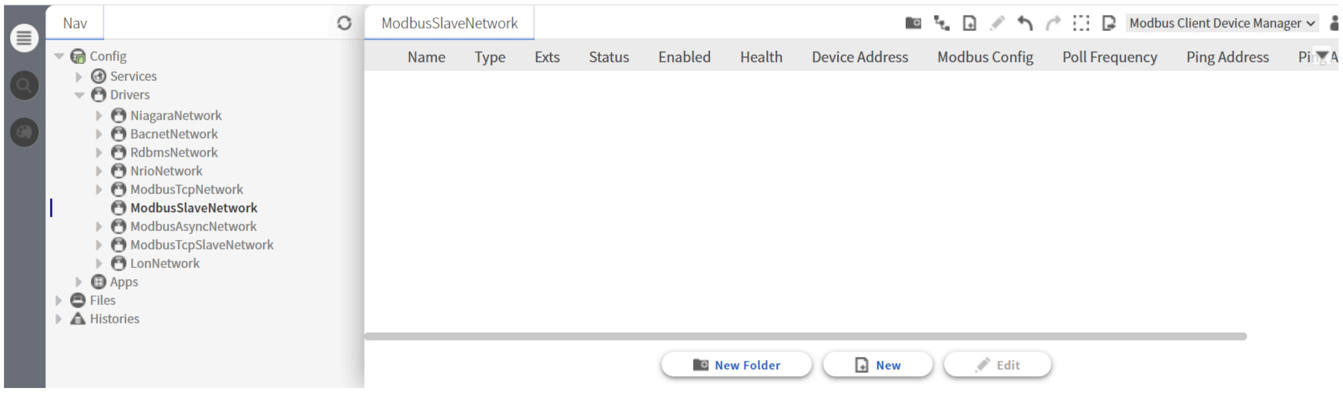
Toolbar for Modbus Tcp Client Device Ux Manager

Toolbar Icon and Name	Description
New Folder	It creates a new folder in the database pane.
Trace	Displays all descendants or immediate children of the selected parent proxy point.
Create new objects	It creates new object in the database pane.
Edit objects	Opens the device’s database record for updating.
Undo	Reverses the previous command.
Redo	Restores a command action after the Undo command has removed it.
Multi-selection Mode	Enables you to individually select multiple points without holding down the ctrl key.
Export	Exports the current view or object.

HTML5- Modbus Client Device Ux Manager

In Niagara 4.14 and later, there is added browser support for Modbus Client Device Manager View. The HTML 5 version of this view is a web-browser-based implementation and it provides the same functions as the Workbench view.

Figure 40. ModbusClientDeviceUxManager



To access this view, expand **Config** > **Drivers** and double-click **ModbusSlaveNetwork** or right-click **ModbusSlaveNetwork** > **Views** > **Modbus Client Device Manager**.

Columns

Column Name	Description
Name	Reports the name of the entity or logical grouping.
Type	Displays the type of database.
Exts	Displays the device extension's hyperlinks, including: Points, Alarms, Schedules, Trend Logs and Config.
Enabled	Indicates if the network, device, point or component is active or inactive.
Status	Reports the current condition of the entity as of the last refresh: {alarm}, {disabled}, {down}, {fault}, {ok}, {stale}, {unackedAlarm}
Health	Displays the status of the network, device or component.
Device Address	Displays the unique number that identifies the current device object on the network
Modbus Config	Displays the settings for message transactions to (and from) only that device.

Buttons

- **New Folder** creates a new folder for devices. Each such folder provides its own set of manager views.
- **New** creates a new device record in the database.
- **Edit** opens the device's database record for updating.

Toolbar for Modbus Client Device Ux Manager

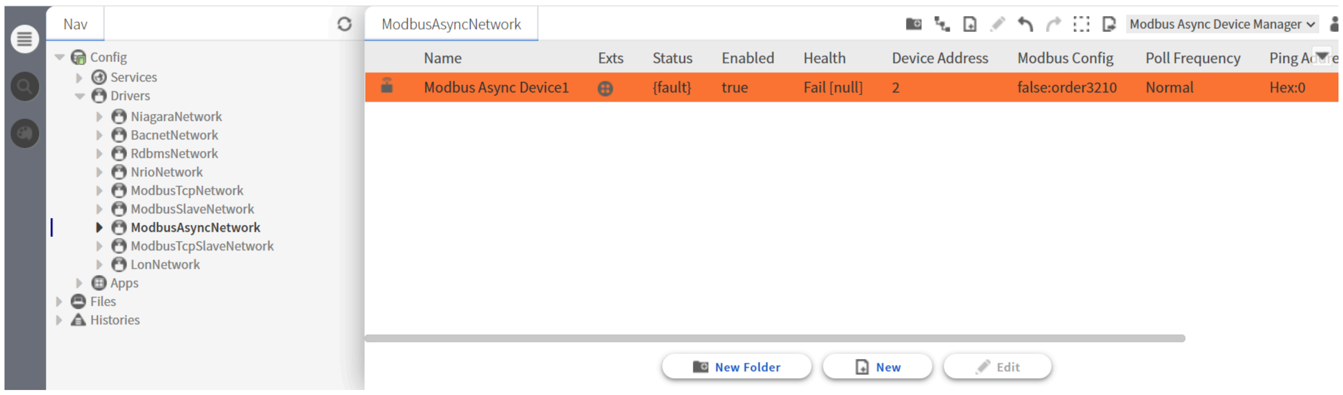
Toolbar Icon and Name	Description
New Folder	It creates a new folder in the database pane.
Trace	Displays all descendants or immediate children of the selected parent proxy point.

Toolbar Icon and Name	Description
Create new objects	It creates new object in the database pane.
Edit objects	Opens the device’s database record for updating.
Undo	Reverses the previous command.
Redo	Restores a command action after the Undo command has removed it.
Multi-selection Mode	Enables you to individually select multiple points without holding down the ctrl key.
Export	Exports the current view or object.

HTML5- Modbus Async Device Ux Manager

In Niagara 4.14 and later, there is added browser support for Modbus Async Device Manager View. The HTML 5 version of this view is a web-browser-based implementation and it provides the same functions as the Workbench view.

Figure 41. ModbusAsyncUxDeviceUxManager



Name	Exts	Status	Enabled	Health	Device Address	Modbus Config	Poll Frequency	Ping A
Modbus Async Device1		{fault}	true	Fail [null]	2	false:order3210	Normal	Hex:0

To access this view, expand **Config** > **Drivers** and double-click **ModbusAsyncNetwork** or right-click **ModbusAsyncNetwork** > **Views** > **Modbus Async Device Manager**.

Columns

Column Name	Description
Name	Reports the name of the entity or logical grouping.
Exts	Displays the device extension's hyperlinks, including: Points, Alarms, Schedules, Trend Logs and Config.
Enabled	Indicates if the network, device, point or component is active or inactive.
Status	Reports the current condition of the entity as of the last refresh: {alarm}, {disabled}, {down}, {fault}, {ok}, {stale}, {unackedAlarm}
Health	Displays the status of the network, device or component.
Device Address	Displays the unique number that identifies the current device object on the network
Modbus Config	Displays the settings for message transactions to (and from) only that device.
Poll Frequency	Displays the poll services that is used in the device
Ping Address	Displays the ping address of the device.
Ping Address Data Type	Displays the data type.

Buttons

- **New Folder** creates a new folder for devices. Each such folder provides its own set of manager views.
- **New** creates a new device record in the database.
- **Edit** opens the device's database record for updating.

Toolbar for Modbus Async Device Ux Manager

Toolbar Icon and Name	Description
New Folder	It creates a new folder in the database pane.
Trace	Displays all descendants or immediate children of the selected parent proxy point.
Create new objects	It creates new object in the database pane.
Edit objects	Opens the device's database record for updating.
Undo	Reverses the previous command.
Redo	Restores a command action after the Undo command has removed it.
Multi-selection Mode	Enables you to individually select multiple points without holding down the ctrl key.

Toolbar Icon and Name	Description
Export	Exports the current view or object.

Chapter 9. Windows

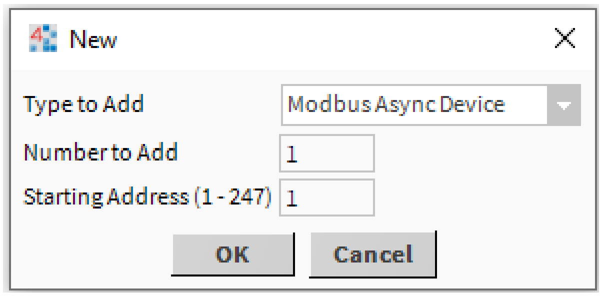
Windows create and edit database records or collect information when accessing a component. You access them by dragging a component from a palette to a Nav tree node or by clicking a button.

Windows do not support **On View (F1)** and **Guide on Target** help. To learn about the information each contains, search the help system for key words.

New device type window-Modbus

This window adds a sequentially-addressed range of multiple Modbus devices.

Figure 42. Example of a New ModbusAsyncDevice window


New

Type to Add: Modbus Async Device

Number to Add: 1

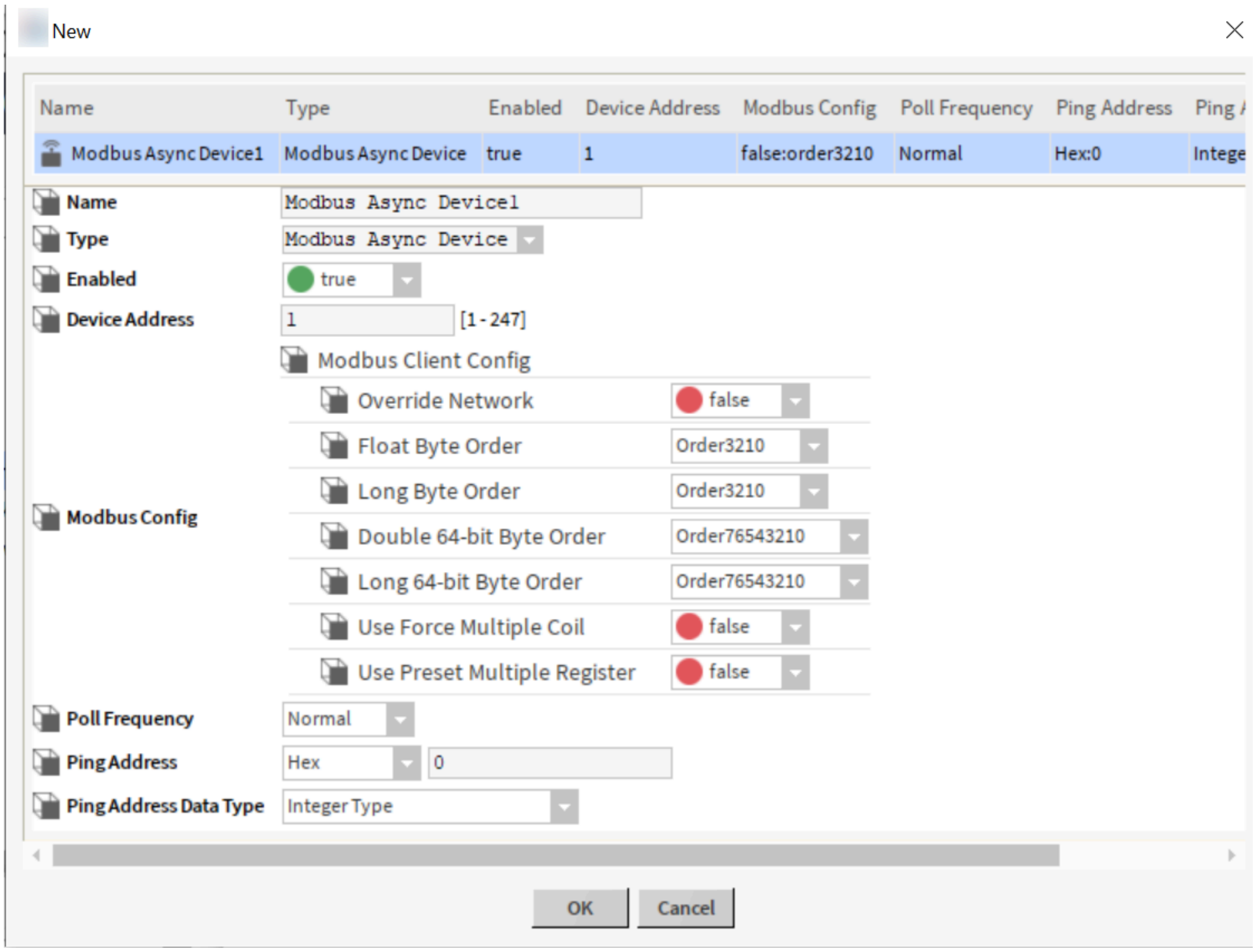
Starting Address (1 - 247): 1

Property	Value	Description
Type to Add	drop-down list	Identifies the network to which the device is connected.
Number to Add	number (defaults to `1`)	Configures how many devices to add.
Starting Address	number between 1 and 247	Defines the first unique device number to assign in a group of Modbus devices.

New device properties window

This window contains the properties for creating a new device.

Figure 43. New device window



Name	Type	Enabled	Device Address	Modbus Config	Poll Frequency	Ping Address	Ping A
Modbus Async Device1	Modbus Async Device	true	1	false:order3210	Normal	Hex:0	Intege

Name: Modbus Async Device1
Type: Modbus Async Device
Enabled: true
Device Address: 1 [1 - 247]
Modbus Config: Modbus Client Config
- Override Network: false
- Float Byte Order: Order3210
- Long Byte Order: Order3210
- Double 64-bit Byte Order: Order76543210
- Long 64-bit Byte Order: Order76543210
- Use Force Multiple Coil: false
- Use Preset Multiple Register: false

Poll Frequency: Normal
Ping Address: Hex 0
Ping Address Data Type: Integer Type

Property	Value	Description
Name	text	Specifies the name of the object.
Type	drop-down list	Specifies the type of the object.
Enabled	`true` or `false` (defaults to `true`)	Activates (`true`) and deactivates (`false`) use of the object (network, device, point, component, table, schedule, descriptor, etc.).
Device Address	number from 1 to 247	Defines the unique number that identifies the current device object on the network.
Modbus Config	additional properties	Refer to Modbus Config .

Property	Value	Description
Poll Frequency	drop-down list	Configures how frequently the system polls proxy points.
Ping Address	additional properties	Defines the base address to use.
Ping Address Data Type	drop-down list	Defines the type of numeric data.

Modbus Config

Each Modbus client device has a **Modbus Config** container slot with five properties. You access these on the device's property sheet, as well as the **New** or **Edit** windows for a device object when in the device manager view of the parent network.

The screen capture shows the properties in the **New** window for a device. These properties allow you to override the network-level (global) Modbus Config equivalent settings for handling Modbus data from and to this device.

Property	Value	Description
Override Network	`true` or `false` (default)	Determines which values to use for these properties: **`Float Byte Order`**, **`Long Byte Order`** and **`Use Force Multiple Coil`**. `false` selects the network-level values as configured by the ModbusAsyncNetwork component. `true` selects the values defined by the **Modbus Config** container slot.
Float Byte Order	drop-down list	Specifies the byte-order used to assemble or receive floating-point (32-bit) values in messages. Choices reflect two alternate methods, where numerals 0, 1, 2, and 3 represent the least significant byte to most significant byte: • `Order3210` – Most significant byte first, or big-endian, it is the default. • `Order1032` – Bytes transmitted in a 1,0,3,2 order, or little–endian. • `Order0123` – Bytes transmitted in a 0,1,2,3 order, or little-endian.
Long Byte Order	drop-down list	Specifies the byte-order used to assemble or receive long integer (32-bit) values in messages.

Property	Value	Description
		Choices reflect two alternate methods, where numerals 0, 1, 2, and 3 represent the least significant byte to most significant byte: • `Order3210` - Most significant byte first, or big-endian, it is the default. • `Order1032` - Bytes transmitted in a 1,0,3,2 order, or little-endian. • `Order0123` - Bytes transmitted in a 0,1,2,3 order, or little-endian. **NOTE:** Float or long values received in incorrect byte order may appear abnormally big, or not at all.
Double 64-bit Byte Order	drop-down list	Specifies the byte-order used to assemble or receive receives double-precision floating point (64-bit) values in messages. Choices reflect in 8 corresponding byte order options, where numerals 0, 1, 2, and 3 represent the least significant byte to most significant byte. When selecting the byte order options, it is required that the Float Byte Order and Long Byte Order are both set to `Order0123` to implement this configuration effectively: • `Order76543210` – Most significant byte first, or big-endian order where the most significant byte is transmitted first (from 7 down to 0). • `Order 67452301` – Bytes transmitted in a order 6,7,4,5, etc or big-endian format involves swapping the bytes. • `Order54761032` – Bytes transmitted in a order 5, 4, 7, 6, etc or big-endian format but with each pair of

Property	Value	Description
		bytes swapped (e.g., switching the first two bytes and the next two bytes). • `Order45670123` – Bytes are transmitted in order 4, 5, 6, 7, etc or big-endian with both words and bytes swapped. • `Order01234567` – Bytes are transmitted in order 0, 1, 2, 3, etc or little-endian (LE) arrangement where the bytes are transmitted in direct order from least significant to most significant (from 0 to 7). • `Order10325476` – Bytes are transmitted in order 1, 0, 3, 2, etc or little-endian format where the bytes are swapped during transmission. • `Order23016745` – Bytes are transmitted in order 2, 3, 0, 1, etc or little-endian format but with each pair of bytes swapped (e.g., switching the first two bytes and the next two bytes). • `Order32107654` – Bytes are transmitted in order 3, 2, 1, 0, etc or little-endian format, with both words and bytes swapped.
Long 64-bit Byte Order	drop-down list	Specifies the byte-order used to assemble or receive long integer (64-bit) values in messages. Choices reflect in 8 corresponding byte order options, where numerals 0, 1, 2, and 3 represent the least significant byte to most significant byte. When selecting the byte order options, it is required that the Float Byte Order and Long Byte Order are both set to `Order0123` to implement this configuration effectively: • `Order76543210` – Most

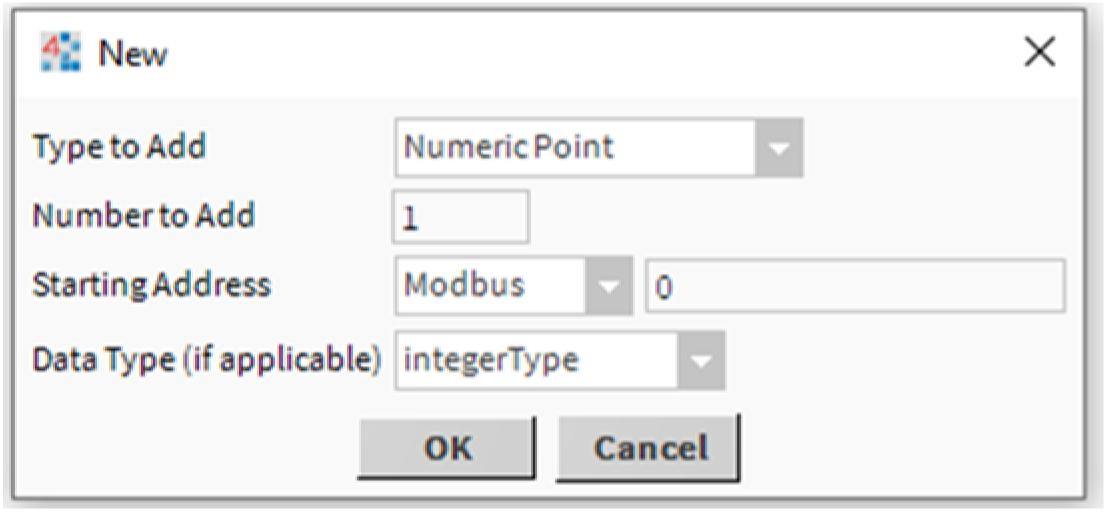
Property	Value	Description
		significant byte first, or big-endian (BE) order where the most significant byte is transmitted first (from 7 down to 0). • `Order 67452301` – Bytes transmitted in a order 6,7,4,5, etc or big-endian format involves swapping the bytes. • `Order54761032` – Bytes transmitted in a order 5, 4, 7, 6, etc or big-endian format but with each pair of bytes swapped (e.g., switching the first two bytes and the next two bytes). • `Order45670123` – Bytes are transmitted in order 4, 5, 6, 7, etc. or big-endian with both words and bytes swapped. • `Order01234567` – Bytes are transmitted in order 0, 1, 2, 3, etc or little-endian (LE) arrangement where the bytes are transmitted in direct order from least significant to most significant (from 0 to 7). • `Order10325476` – Bytes are transmitted in order 1, 0, 3, 2, etc or little-endian format where the bytes are swapped during transmission. • `Order23016745` – Bytes are transmitted in order 2, 3, 0, 1, etc or little-endian format but with each pair of bytes swapped (e.g., switching the first two bytes and the next two bytes). • `Order32107654` – Bytes are transmitted in order 3, 2, 1, 0, etc or little-endian format, with both words and bytes swapped.
Use Force Multiple Coil	`true` or `false` (default)	Specifies whether to use function code 15 (Force Multiple Coils)

Property	Value	Description
		instead of function code 05 (Force Single Coil) when writing to coils. The default is `false`, where function code 05 (Force Single Coil) is used. This property depends on the Modbus function codes supported by child devices, which (if available) provide alternative options in Modbus messaging. Function code 15 support (Preset Multiple Coils) is available in devices (true or false). The default is `false`, where function code **`Preset Single Coil`** is in place.
Use Preset Multiple Register	`true` or `false` (default)	Specifies whether to use function code 16 (**`Preset Multiple Registers`**) instead of function code 06 (**`Preset Single Register`**) when writing to registers. The default is `false`, where function code 06 (**`Preset Single Register`**) is used. This property depends on the Modbus function codes supported by child devices, which (if available) provide alternative options in Modbus messaging. Function code 16 support (**`Preset Multiple Registers`**) is available in devices (`true` or `false`). The default is `false`, where function code **`Preset Single Register`** is in place.

New point type window

This window creates a new proxy point.

Figure 44. New point type properties



Type to Add: Numeric Point
Number to Add: 1
Starting Address: Modbus 0
Data Type (if applicable): integerType

You access these properties from the point manager view by clicking the **New** button.

Property	Value	Description
Type to Add	drop-down list	Selects the type of point.
Number to Add	number	Selects the number of instances of point.
Starting Address	number between 1 and 247	Defines the first unique device number to assign in a group of Modbus devices.
Data Type	drop-down list	Selects the location of the data in the slave device. Coils store on/off values. Registers store numeric values. Each coil or contact is 1 bit with an assigned address between 0000 and 270E. Each register is one word (16 bits, 2 bytes) as well as a data address between 0000 and 270E.

Modbus proxy points

Modbus client and server proxy points are similar to other driver proxy points. The same collection of client proxy points is used in devices under a ModbusAsync, ModbusTcp, ModbusTcpGateway, ModbusSlave and ModbusTcpSlave network components. You can find them in the client **Points** folder in the palettes (**modbusAsync**, **modbusTcp**, **modbusSlave** and **modbusTcpSlave**).

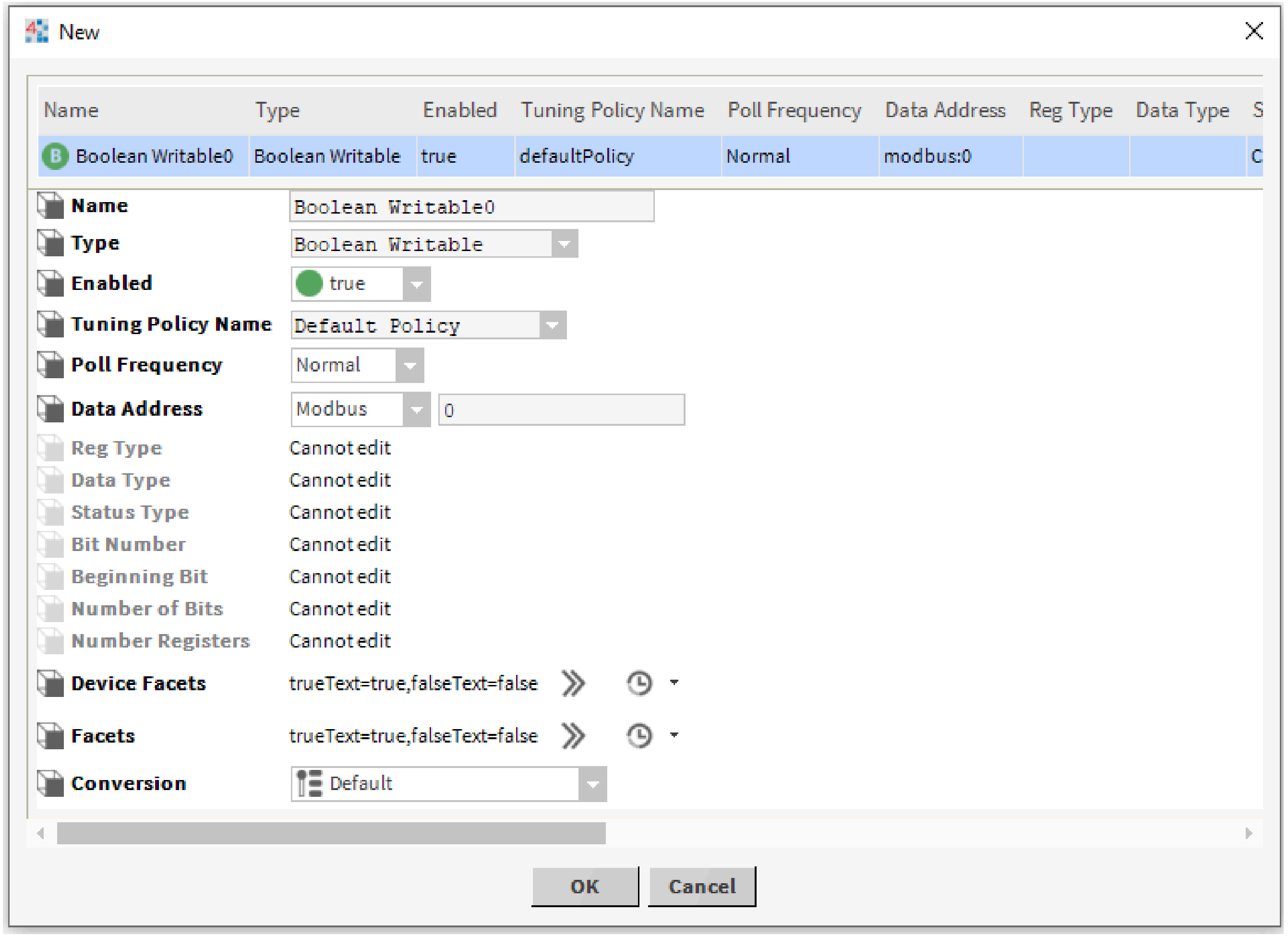
Proxy point type	Client (master) usage	Server (slave) usage
Boolean Writable	Reads and writes a Modbus coil.	Reads and writes a virtual Modbus coil or input. Generally, it is unwise to expose any coil as

Proxy point type	Client (master) usage	Server (slave) usage
		BooleanWritable if the Modbus master may also write to this same item—otherwise write contention issues may result.
Boolean Point	Reads either a Modbus coil or an input.	Reads a virtual Modbus coil that may be written by the Modbus master.
Numeric Writable	Reads and writes a Modbus holding register value. You must specify the **`Data Type`** as either integer, long, float, or signed integer.	Reads and writes a virtual Modbus holding register value or input register value. You must specify the **`Data Type`** as either integer, long, float, or signed integer. Generally, it is unwise to expose any holding register as NumericWritable if the Modbus master may also write to this same item—otherwise write contention issues may result.
Numeric Point	Reads either a Modbus holding register value or an input register value. You must specify the **`Data Type`** as either integer, long, float, or signed integer.	Reads a virtual Modbus holding register value that may be written by the Modbus master device. You must specify the Data Type as either integer, long, float, or signed integer.
Register Bit Writable	Reads and writes a specific bit in a Modbus holding register (select Bit Number in setup).	Reads and writes a specific bit in a virtual Modbus holding register or input register (select Bit Number in setup). Generally, it is unwise to expose any holding register as a RegisterBitWritable if the Modbus master may also write to this same item—otherwise write contention issues may result.
Register Bit Point	Reads a specific bit in either a Modbus holding register or an input register (select Bit Number in setup).	Reads a specific bit in a virtual Modbus holding register (select Bit Number in setup) that may be written by the Modbus master.
String point	Reads some number of consecutive Modbus holding registers and interpret them as an ASCII string, using a high-to-low byte order. In general, use of this type is expected to be infrequent.	Reads some number of consecutive virtual Modbus holding registers that may be written by the Modbus master, and interpret them as an ASCII string, using a high-to-low byte order. In general, use of this type is expected to be infrequent.
Enum Bits Writable	Reads and writes some number of consecutive bits within a holding register, with the resulting integer as the out (ordinal) value of the Enum Writable.	N/A
Enum Bits Point	Reads some number of consecutive bits within a holding or input registering the resulting integer as the out (ordinal) value of the Enum Point.	N/A
Numeric Bits Writable	Reads and writes some number of consecutive bits within a holding registering the resulting integer as the out value of the Numeric Writable.	N/A
Numeric Bits Point	Reads some number of consecutive bits within a holding or input registering the resulting integer as the out value of the Numeric Point.	N/A

New point properties window

This window configures the point properties.

Figure 45. New point properties



Name	Type	Enabled	Tuning Policy Name	Poll Frequency	Data Address	Reg Type	Data Type	S
Boolean Writable0	Boolean Writable	true	defaultPolicy	Normal	modbus:0			C

Name	Boolean Writable0
Type	Boolean Writable
Enabled	true
Tuning Policy Name	Default Policy
Poll Frequency	Normal
Data Address	Modbus 0
Reg Type	Cannot edit
Data Type	Cannot edit
Status Type	Cannot edit
Bit Number	Cannot edit
Beginning Bit	Cannot edit
Number of Bits	Cannot edit
Number Registers	Cannot edit
Device Facets	trueText=true,falseText=false
Facets	trueText=true,falseText=false
Conversion	Default

Property	Value	Description
Name	text	Shows the name of the point as reported by the device.
Type	drop-down list	Selects the type of point.
Enabled	`true` or `false` (defaults to `true`)	Activates (`true`) and deactivates (`false`) use of the object (network, device, point, component, table, schedule, descriptor, etc.).
Tuning Policy Name	drop-down list	Selects a network tuning policy by name. This policy defines stale time and minimum and maximum update times.

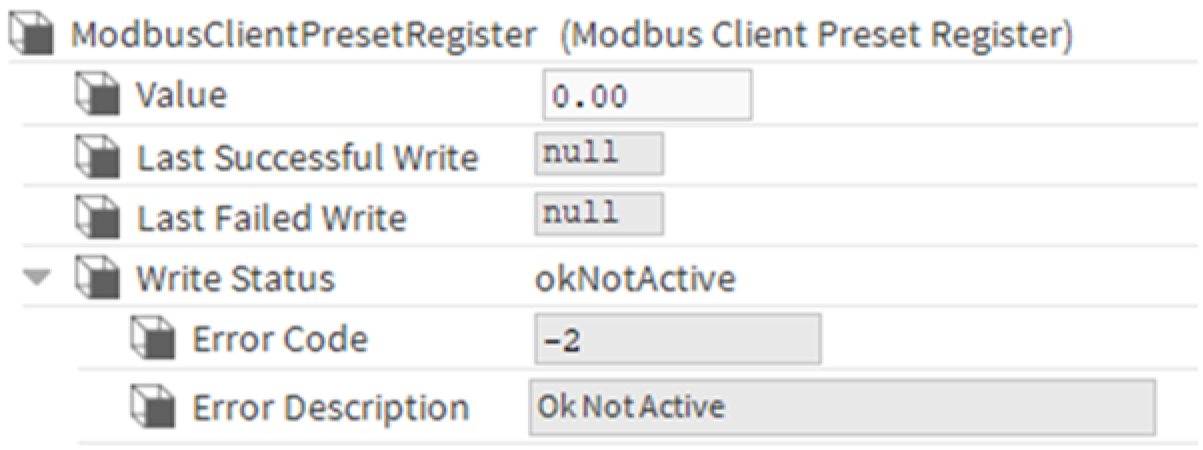
Property	Value	Description
		During polling, the system uses the tuning policy to evaluate both write requests and the acceptability (freshness) of read requests.
Poll Frequency	drop-down list	Configures how frequently the system polls proxy points.
Facets-Boolean	*trueText* (default) or *falseText*	Define the text to display for the Boolean values: • `trueText` is the text to display when output is true • `falseText` is the text to display when output is false. For example, the facet `trueText` could display "ON" and the facet `falseText` "OFF." You view Facets on the Slot Sheet and edit them from a component Property Sheet by clicking the >> icon to display the Config Facets window.
Conversion	drop-down list	Selects the units to use when converting values from the device facets to point facets. `Default` automatically converts similar units (such as Fahrenheit to Celsius) within the proxy point. **NOTE:** In most cases, the standard `Default` is best. `Linear` applies to voltage input, resistive input and voltage output writable points. Works with linear-acting devices. You use the Scale and Offset properties to convert the output value to a unit other than that defined by device facets. `Linear With Unit` is an extension to the existing linear conversion property. This specifies whether the unit conversion should occur

Property	Value	Description
		on “Device Value” or “Proxy Value”. The new linear with unit convertor, will have a property to indicate whether the unit conversion should take place before or after the scale/offset conversion. `Reverse Polarity` applies only to Boolean input and relay output writable points. Reverses the logic of the hardware binary input or output. `500 Ohm Shunt` applies to voltage input points only. It reads a 4-to-20mA sensor, where the Ui input requires a 500 ohm resistor wired across (shunting) the input terminals. `Tabular Thermistor` applies to only a Thermistor input point and involves a custom resistance-to-temperature value response curve for Type 3 Thermistor temperature sensors. `Thermistor Type 3` applies to an Thermistor Input point, where this selection provides a “built-in” input resistance-to-temperature value response curve for Type 3 Thermistor temperature sensors. `Generic Tabular` applies to non-linear support for devices other than for thermistor temperature sensors with units in temperature. Generic Tabular uses a lookup table method similar to the “Thermistor Tabular” conversion, but without predefined output units.

Add Preset Register Value window

This window configures a Modbus Client Preset Register.

Figure 46. Add Preset Register Value properties

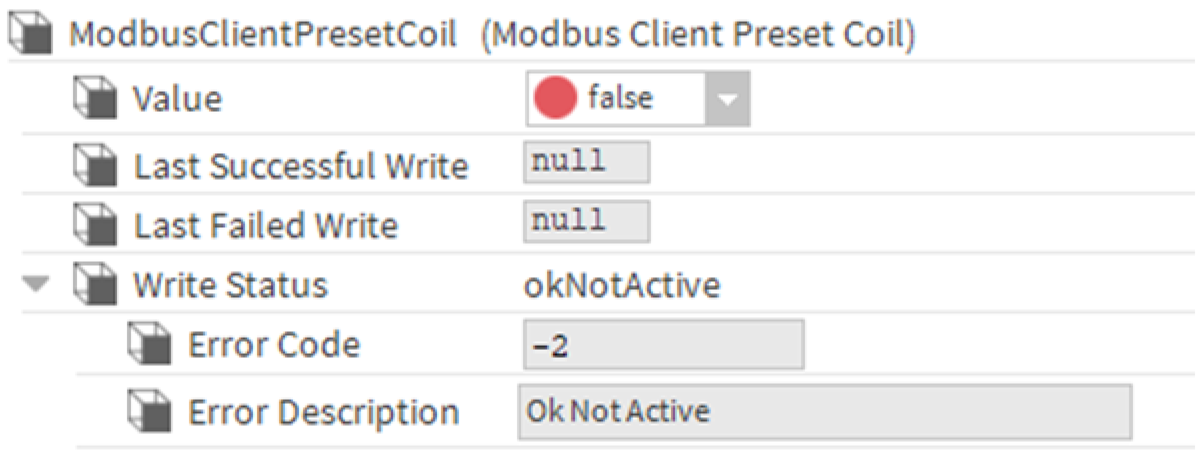

ModbusClientPresetRegister (Modbus Client Preset Register)

Value	0.00
Last Successful Write	null
Last Failed Write	null
Write Status	okNotActive
Error Code	-2
Error Description	Ok Not Active

Property	Value	Description
Value	number to two decimal places (defaults to 0.00)	Configures the value of the register.
Last Successful Write	read-only date and time	Reports the last successful write.
Last Failed Write	read-only date and time	Reports the last failed write.
Write Status	Additional properties	Reports if the object is read-only or can be written to.

Add Preset Coil Value window

This window configures a Modbus Client Preset Coil.


ModbusClientPresetCoil (Modbus Client Preset Coil)

Value	● false
Last Successful Write	null
Last Failed Write	null
Write Status	okNotActive
Error Code	-2
Error Description	Ok Not Active

Figure 47. Add Preset Coil Value properties

Property	Value	Description
Value	number	Configures the value of the register.
Last Successful Write	read-only date and time	Reports the last successful write.
Last Failed Write	read-only date and time	Reports the last failed write.
Write Status	Additional properties	Reports if the object is read-only or can be written to.