

Technical Document

Niagara^{AX-3.x} Modbus Guide

Updated: August 25, 2010



Niagara^{AX} Modbus Guide

Confidentiality Notice

The information contained in this document is confidential information of Tridium, Inc., a Delaware corporation (“Tridium”). Such information, and the software described herein, is furnished under a license agreement and may be used only in accordance with that agreement.

The information contained in this document is provided solely for use by Tridium employees, licensees, and system owners; and, except as permitted under the below copyright notice, is not to be released to, or reproduced for, anyone else.

While every effort has been made to assure the accuracy of this document, Tridium is not responsible for damages of any kind, including without limitation consequential damages, arising from the application of the information contained herein. Information and specifications published here are current as of the date of this publication and are subject to change without notice. The latest product specifications can be found by contacting our corporate headquarters, Richmond, Virginia.

Trademark Notice

Modbus is a registered trademark of Schneider Electric, Inc. BACnet and ASHRAE are registered trademarks of American Society of Heating, Refrigerating and Air-Conditioning Engineers. Microsoft and Windows are registered trademarks, and Windows NT, Windows 2000, Windows XP Professional, and Internet Explorer are trademarks of Microsoft Corporation. Java and other Java-based names are trademarks of Sun Microsystems Inc. and refer to Sun's family of Java-branded technologies. Mozilla and Firefox are trademarks of the Mozilla Foundation. Echelon, LON, LonMark, LonTalk, and LonWorks are registered trademarks of Echelon Corporation. Tridium, JACE, Niagara Framework, Niagara^{AX} Framework, and Sedona Framework are registered trademarks, and Workbench, WorkPlace^{AX}, and ^{AX}Supervisor, are trademarks of Tridium Inc. All other product names and services mentioned in this publication that is known to be trademarks, registered trademarks, or service marks are the property of their respective owners.

Copyright and Patent Notice

This document may be copied by parties who are authorized to distribute Tridium products in connection with distribution of those products, subject to the contracts that authorize such distribution. It may not otherwise, in whole or in part, be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine-readable form without prior written consent from Tridium, Inc.

Copyright © 2010 Tridium, Inc.

All rights reserved. The product(s) described herein may be covered by one or more U.S or foreign patents of Tridium.

CONTENTS

Preface	v
About this document	1-v
Modbus terms	1-v
Document change log	vi
Modbus Driver Installations	1-1
Modbus license requirements	1-1
Installing Modbus software	1-1
Modbus Quick Start	2-1
Configure the client Modbus network	2-1
Add a ModbusAsyncNetwork	2-1
Configure the serial port parameters and transmission mode	2-2
Add a ModbusTcpNetwork	2-2
Add a ModbusTcpGateway	2-2
Review network-level Modbus client configuration	2-3
Add client Modbus devices	2-3
Create Modbus client proxy points	2-4
Add other client Modbus components	2-5
Configure the slave Modbus network	2-6
Add a ModbusSlaveNetwork	2-6
Configure the serial port parameters and transmission mode	2-6
Add a ModbusTcpSlaveNetwork	2-7
Review network-level Modbus server configuration	2-7
Add server (slave) Modbus devices	2-8
Create Modbus server proxy points	2-9
Add other server Modbus components	2-9
Modbus Meets NiagaraAX	3-1
Brief History	3-1
Modbus key concepts	3-1
Modbus registers	3-2
Modbus data addresses	3-2
Organization and addressing of data	3-2
Consecutive addresses	3-3
Consecutive address usage (general)	3-3
Consecutive address usage (NiagaraAX)	3-4
Data address format in NiagaraAX	3-5
Modbus data types	3-6
Data in Modbus	3-6
Numerical data types	3-6
NiagaraAX data representation	3-7

Modbus function codes	3-7
<i>Supported function codes</i>	3-7
Modbus messages	3-8
<i>Message structures</i>	3-8
<i>NiagaraAX debug example</i>	3-9
Exception responses	3-9
<i>Exception response format</i>	3-9
<i>Exception codes</i>	3-10
Device-to-device differences	3-11
Network-level settings	3-11
Device-level Modbus Config properties	3-11

NiagaraAX Modbus Representation 4-1

About Modbus Async networks	4-1
Modbus Async Network configuration	4-2
Modbus Async Device Manager notes	4-3
About Modbus TCP networks	4-4
Modbus Tcp Network configuration	4-4
Modbus Tcp Device Manager notes	4-5
About Modbus TCP Gateway networks	4-6
Modbus Tcp Gateway configuration	4-7
Modbus Tcp Gateway Device Manager notes	4-8
About Modbus Slave networks	4-8
Modbus Slave Network configuration	4-9
Modbus Slave Device Manager notes	4-10
About Modbus TCP Slave networks	4-10
Modbus Tcp Slave Network configuration	4-11
Modbus Tcp Slave Device Manager notes	4-11
About Master (client) types	4-12
About Modbus client devices	4-12
<i>Modbus Config (client device level)</i>	4-12
<i>Ping Address properties</i>	4-13
<i>Base Address properties</i>	4-14
<i>Device Poll Config</i>	4-15
Modbus Client Point Manager notes	4-16
About Modbus client proxy points	4-17
<i>Types of Modbus client proxy points</i>	4-18
<i>Modbus client point ProxyExt properties</i>	4-19
About Modbus preset components	4-21
<i>Modbus Client Preset Coils</i>	4-22
<i>Modbus Client Preset Registers</i>	4-23
<i>Adding client presets</i>	4-24
About Modbus (client) file records	4-25
About Modbus client exception status	4-25
About Slave (server) types	4-26
About Modbus server devices	4-26
<i>Modbus Config (server device level)</i>	4-26
<i>Modbus Register Range Tables</i>	4-27
About Modbus server proxy points	4-29
<i>Types of Modbus server proxy points</i>	4-29
<i>Modbus server point ProxyExt properties</i>	4-30
About Modbus (server) file records	4-31

Modbus Plugin Guides 5-1

Plugin Guides Summary	5-1
modbusAsync plugins	5-1
modbusCore plugins	5-1

modbusSlave plugins 5-2
modbusTcp plugins 5-2
modbusTcpSlave plugins 5-2

Modbus Component Guides 6-1

Component Reference Summary 6-1
modbusAsync components 6-1
modbusCore Components 6-2
modbusSlave components 6-5
modbusTcp components 6-6
modbusTcpSlave components 6-7

PREFACE

Preface

- [About this document](#)
- [Modbus terms](#)
- [Document change log](#)

About this document

This document applies to any of the Modbus drivers for any release of NiagaraAX. Any features that require a certain release level of NiagaraAX are noted, if necessary.

Note: *The audience is considered to be knowledgeable in Modbus technology and is NiagaraAX Certified.*

The following main sections are included:

- This preface.
Includes [Modbus terms](#) and a [Document change log](#).
- [“Modbus Driver Installations”](#) on page 1-1
Explains NiagaraAX license requirements and the software modules that need to be installed.
- [“Modbus Quick Start”](#) on page 2-1
Provides several quick procedures for station configuration to add various Modbus client or slave networks, add child Modbus devices and proxy points, and also other Modbus components.
- [“Modbus Meets NiagaraAX”](#) on page 3-1
Provides general explanations of basic Modbus concepts. Included is basic information on registers, addressing, data types, function codes, device differences, and Modbus messaging.
- [“NiagaraAX Modbus Representation”](#) on page 4-1
Provides details on NiagaraAX components and views used to model Modbus devices and networks, including the different client types and server types. Included are details on Modbus proxy points.
- [“Modbus Plugin Guides”](#) on page 5-1
Provides brief summaries of the different Modbus manager views, each with links back to the more detailed concepts section. Entries are used in NiagaraAX context-sensitive help “On View”.
- [“Modbus Component Guides”](#) on page 6-1
Provides brief summaries of the different Modbus components, most with links back to the more detailed concepts section. Entries are used in NiagaraAX context-sensitive help “Guide On Target”.

Modbus terms

The following list of terms and abbreviations is specific to Modbus usage in NiagaraAX, and covers entries used in this document. For general NiagaraAX terms, see the Glossary in the *User Guide*.

ASCII American Standard Code for Information Interchange (or in frequent context), Modbus ASCII. One of two Modbus serial transmission modes, where two eight-bit bytes of information are sent as two ASCII characters. Typically, most Modbus serial devices use the Modbus RTU protocol instead.

coils Discrete “On/Off” outputs in a Modbus slave that can be read and typically written by the Modbus master. One of four different Modbus data groups. The term “coil” originated from the first PLC applications, in which outputs of a PLC were set by energizing coils of output relays. See [“Modbus registers”](#) on page 3-2.

COV Change-of-Value.

CRC Cyclic Redundancy Check. An error checksum mechanism used in Modbus RTU.

exception code A numerical code (contained within an exception response) that explains why a successful response to a query cannot be delivered. See “[Exception codes](#)” on page 3-10.

exception response A response sent by a Modbus slave when the query message (sent by the master) cannot be successfully delivered, for some reason. See “[Exception responses](#)” on page 3-9.

function code One of numerous functions defined within the Modbus specification, of which a device may implement. Each function has a numerical code. See “[Modbus function codes](#)” on page 3-7.

holding registers 16-bit (2-byte) data registers in a Modbus slave that hold values that can typically be read and written by the Modbus master. Values may use different data types, such as integer, float, long, and others. One of four different Modbus data groups. See “[Modbus registers](#)” on page 3-2.

hex For hexadecimal. The base-16 numerical format used to describe Modbus message transactions, and sometimes used for Modbus data addressing.

inputs Relating to Modbus, refers to the discrete “On/Off” status of digital inputs for a Modbus slave. Can be read (only) by the Modbus master. One of four different Modbus data groups. See “[Modbus registers](#)” on page 3-2.

input registers 16-bit (2-byte) data registers in a Modbus slave that hold read-only values. Values may use different data types, such as integer, float, long, or others. One of four different Modbus data groups. See “[Modbus registers](#)” on page 3-2.

query In general Modbus terms, a message sent from the Modbus master to a slave, to retrieve or write a value in a data item. See “[Modbus messages](#)” on page 3-8.

register In general Modbus terms, an addressable 2-byte (16-bit) memory location in a Modbus slave that can hold a data value. There are two main types of registers: [input registers](#) and [holding registers](#).

response In general Modbus terms, a message reply from the Modbus slave sent to the master, typically with a requested data value or other confirmation. See “[Modbus registers](#)” on page 3-2.

RTU Remote Terminal Unit, or simply Modbus RTU. One of two Modbus serial transmission modes, where data is sent as two four-bit, hexadecimal characters. This provides higher throughput than using the (older) Modbus [ASCII](#) protocol for the same serial baud rate.

status Specific to Modbus, status implies boolean (binary or On/Off) data, such as for Modbus data types coils and inputs. As used in Niagara, status can also mean general “health” of an object or output, such as “down,” “fault,” or “ok.”

TCP Transmission Control Protocol (or in frequent context) Modbus TCP. An open Modbus protocol that facilitates Modbus message transfer using TCP/IP protocol and standard Ethernet networks.

Document change log

Updates (changes/additions) to this *NiagaraAX Modbus Guide* document are listed below.

- Updated: August 25, 2010
Added “[About this document](#)” section in the this Preface. Added details about client Modbus “Enum Bits” and “Numeric Bits” proxy points made available in build 3.5.26 or later (or AX-3.6 or higher), which affected sections “[NiagaraAX data representation](#)” on page 3-7, “[Types of Modbus client proxy points](#)” on page 4-18, and in reference topics for related proxy extensions [ModbusClientEnumBitsProxyExt](#) and [ModbusClientNumericBitsProxyExt](#). Added details about client component to read function code 07 (exception status), which affected section “[Supported function codes](#)” on page 3-7, along with new sections “[About Modbus client exception status](#)” on page 4-25, and summary reference [ModbusClientExceptionStatus](#). Updated document with newer style legal text.
- Updated: February 14, 2008
Changed document to reference the new *NiagaraAX Drivers Guide*, which was created from information formerly found in the *NiagaraAX User Guide*.
- Updated: March 15, 2007
Completely reworked what was formerly a “placeholder” document. Included are new main sections “[Modbus Quick Start](#)”, “[Modbus Meets NiagaraAX](#)”, and “[NiagaraAX Modbus Representation](#)”, along with numerous content changes in the “[Modbus Plugin Guides](#)” and “[Modbus Component Guides](#)” summary descriptions. Also added was “[Modbus terms](#)” in this Preface.
- Published: June 24, 2005
Initial document.

CHAPTER 1

Modbus Driver Installations

Currently, this section has only two subsections:

- [Modbus license requirements](#)
- [Installing Modbus software](#)

Modbus license requirements

To use any of the NiagaraAX Modbus drivers, you must have a target NiagaraAX host (JACE) that is licensed with the corresponding feature(s). These include the following:

- `modbusAsync` — For ModbusAsyncNetworks (serial Modbus [RTU](#) or [ASCII](#) over RS-485 or RS-232)
- `modbusSlave` — For ModbusSlaveNetworks (serial Modbus [RTU](#) or [ASCII](#) over RS-485 or RS-232)
- `modbusTcp` — For ModbusTcpNetworks and/or ModbusTcpGateways (Modbus [TCP](#) via Ethernet)
- `modbusTcpSlave` — For ModbusTcpSlaveNetworks (Modbus [TCP](#) via Ethernet)

In addition, note that other limits on devices and proxy points may exist in your license.

Installing Modbus software

From your PC, use the Niagara Workbench 3.*n.nn* installed with the “installation tool” option (checkbox “This instance of Workbench will be used as an installation tool”). This option installs the needed distribution files (*.dist* files) for commissioning various models of remote JACE platforms. The dist files are located under your Niagara install directory under a “sw” subdirectory.

For details, see “About your software database” in the *Platform Guide*.

Apart from installing the 3.*n.nn* version of the Niagara distribution in the JACE, make sure to also install the *modbusCore* module, plus any specific *modbus<type>* module needed (for example, *modbusAsync*, *modbusTcp*, and so on). Upgrade any modules shown as “out of date”.

For details, see “Software Manager” in the *Platform Guide*.

Following this, the remote JACE is now ready for Modbus configuration in its running station, as described in the rest of this document. See the next section “[Modbus Quick Start](#)” for a series of task-based procedures, as well as other sections “[Modbus Meets NiagaraAX](#)” and “[NiagaraAX Modbus Representation](#)” for conceptual and operational topics.

CHAPTER 2

Modbus Quick Start

This section provides a collection of procedures to use the NiagaraAX Modbus drivers to build networks of devices with proxy points and other components. Like other NiagaraAX drivers, you can do most configuration from special “manager” views and property sheets using Workbench.

These are the main subsections:

- For any of the “client” (master) Modbus networks:
 - [“Configure the client Modbus network”](#) on page 2-1
 - [“Add client Modbus devices”](#) on page 2-3
 - [“Create Modbus client proxy points”](#) on page 2-4
 - [“Add other client Modbus components”](#) on page 2-5
- For either of the “server” (slave) Modbus networks:
 - [“Configure the slave Modbus network”](#) on page 2-6
 - [“Add server \(slave\) Modbus devices”](#) on page 2-8
 - [“Create Modbus server proxy points”](#) on page 2-9
 - [“Add other server Modbus components”](#) on page 2-9

Configure the client Modbus network

To add and configure a client Modbus network, perform the following main tasks:

- Add the client Modbus network, as needed:
 - [Add a ModbusAsyncNetwork](#)
 - [Add a ModbusTcpNetwork](#)
 - [Add a ModbusTcpGateway](#)
- [Review network-level Modbus client configuration](#)

Add a ModbusAsyncNetwork

Note: First see [“Modbus Driver Installations”](#) on page 1-1 for license and software requirements. For background information, see [“About Modbus Async networks”](#) on page 4-1.

To add a ModbusAsyncNetwork in the station

Use the following procedure to add a ModbusAsyncNetwork component under the station’s Drivers container.

Note: If the host JACE has multiple RS-485 or RS-232 ports to be used for client (master) access of Modbus networks, add one ModbusAsyncNetwork for each physical port. Note that for each ModbusAsyncNetwork, you must [Configure the serial port parameters and transmission mode](#).

To add a ModbusAsyncNetwork in the station:

- Step 1 Double-click the station’s **Drivers** container, to bring up the **Driver Manager**.
- Step 2 Click the **New** button to bring up the New network dialog. For more details, see “Driver Manager New and Edit” in the *Drivers Guide*.
- Step 3 Select “Modbus Async Network,” number to add: 1 (or more if multiple networks) and click **OK**. This brings up a dialog to name the network(s).
- Step 4 Click **OK** to add the ModbusAsyncNetwork(s) to the station. You should have a ModbusAsyncNetwork named “ModbusAsyncNetwork” (or whatever you named it), under your Drivers folder, initially showing a status of “{fault}” and enabled as “true.” After you [Configure the serial port parameters and transmission mode](#), status should change to “{ok}”.

Configure the serial port parameters and transmission mode

In the ModbusAsyncNetwork property sheet for each network, you must set the serial port configuration to match the serial communications parameters used by other Modbus devices on the network, including the Modbus transmission mode (RTU or ASCII).

To set the serial port parameters

To set the serial port parameters and mode for a ModbusAsyncNetwork:

- Step 1 Right-click the ModbusAsyncNetwork and select **Views > Property Sheet**. The **Property Sheet** appears.
 - Step 2 Scroll down and expand the **Serial Port Config** slot
Set the properties for the JACE serial port used, where defaults are:
 - Port Name: none — Enter the JACE port being used, like COM2 or COM3.
 - Baud Rate: Baud9600 — Or choose different from selection list.
 - Data Bits: Data Bits8 — Or choose different from selection list.
 - Stop Bits: Stop Bit1 — Or choose different from selection list.
 - Parity: none — Or choose different from selection list.
 - Flow Control Mode: none — Or choose different using checkbox.
- Note:** You must determine the setup of the Modbus serial network to correctly set the baud rate, data bits, stop bits, parity, and flow control settings.
- 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, you should review its "global" Modbus settings. See "[Review network-level Modbus client configuration](#)" on page 2-3.

Add a ModbusTcpNetwork

Note: First see "[Modbus Driver Installations](#)" on page 1-1 for license and software requirements. For background information, see "[About Modbus TCP networks](#)" on page 4-4.

To add a ModbusTcpNetwork in the station

Use the following procedure to add a ModbusTcpNetwork component under the station's Drivers container.

Note: Only one ModbusTcpNetwork is needed, even if the JACE host has two Ethernet ports connected to two different (non-routed) TCP/IP LANs. In this case, the destination IP addresses of child ModbusTcpDevices will automatically determine the Ethernet port utilized.

To add a ModbusTcpNetwork in the station:

- Step 1 Double-click the station's **Drivers** container, to bring up the **Driver Manager**.
- Step 2 Click the **New** button to bring up the New network dialog. For more details, see "Driver Manager New and Edit" in the *Drivers Guide*.
- Step 3 Select "Modbus Tcp Network," number to add: 1 and click **OK**.
This brings up a dialog to name the network.
- Step 4 Click **OK** to add the ModbusTcpNetwork to the station.
You should have a ModbusTcpNetwork named "ModbusTcpNetwork" (or whatever you named it), under your Drivers folder, initially showing a status of "{ok}" and enabled as "true."
- Step 5 Right-click the ModbusTcpNetwork and select **Views > Property Sheet**.
The **Property Sheet** appears. See "[Review network-level Modbus client configuration](#)" on page 2-3.

Add a ModbusTcpGateway

Note: First see "[Modbus Driver Installations](#)" on page 1-1 for license and software requirements. For background information, see "[About Modbus TCP Gateway networks](#)" on page 4-6.

To add a ModbusTcpGateway in the station

Use the following procedure to add a ModbusTcpGateway (network) component under the station's Drivers container.

Note: One or more ModbusTcpGateways are supported. Often, Modbus TCP/serial gateways are on the same LAN as other Modbus TCP devices.

To add a ModbusTcpGateway in the station:

- Step 1 Double-click the station's **Drivers** container, to bring up the **Driver Manager**.
- Step 2 Click the **New** button to bring up the New network dialog. For more details, see "Driver Manager New and Edit" in the *Drivers Guide*.
- Step 3 Select "Modbus Tcp Gateway," number to add: 1 (or more, if multiple) and click **OK**.
This brings up a dialog to name the network(s).
- Step 4 Click **OK** to add the ModbusTcpGateway(s) to the station.
You should have a ModbusTcpGateway named "ModbusTcpGateway" (or whatever you named it), under your Drivers folder, initially showing a status of "{ok}" and enabled as "true."
- Step 5 Right-click the ModbusTcpGateway and select **Views > Property Sheet**.
The **Property Sheet** appears.
- Step 6 In the **Ip Address** property, enter the Modbus gateway's unique IP address, replacing the "###.###.###.###" default value.
- Step 7 In the **Port** property, review the default 502 default value.
This is the "standard" port used by Modbus TCP. If the Modbus TCP/serial gateway is using another TCP port, change this value to match.
- Step 8 Click the **Save** button.
- Step 9 While this network's property sheet is open, you should review its "global" Modbus settings.
See the next section "[Review network-level Modbus client configuration](#)".

Review network-level Modbus client configuration

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

Note: *These settings apply to all Modbus devices in the network, unless overridden in the configuration of any device, using equivalent properties. For more information, see "Device-to-device differences" on page 3-11.*

To review network-level Modbus configuration

To review network-level Modbus configuration:

- Step 1 In the property sheet of the network, review the following properties:
 - Float Byte Order: `Order3210` — Or select `Order1032` instead if used by most devices.
 - Long Byte Order: `Order3210` — Or select `Order1032` instead if used by most devices.
 - Use Preset Multiple Register: `false` — Or set to `true` if most devices support it (function code 16).
 - Use Force Multiple Coil: `false` — Or set to `true` if most devices support it (function code 15).
- Step 2 If you made any changes, click the **Save** button.

Add client Modbus devices

After adding a client Modbus network, you can use the network's default "device manager" view to add the appropriate client Modbus devices.

Note: *You need the address information for each Modbus device you are adding, as well as its Modbus data configuration (coils, inputs, input registers, and holding registers) for this procedure, as well as for later procedures to add proxy points under devices. Have the device vendor's documentation available for reference, in order to map Modbus data correctly in NiagaraAX.*

To add a client Modbus device in the network

Use the following procedure to add the correct type of client Modbus device in the network.

To add a client Modbus device:

- Step 1 In the Nav tree or in the Driver Manager view, double-click the client network, to bring up the device manager (Modbus Async Device Manager, Modbus Tcp Device Manager, Modbus Tcp Gateway Device Manager). All of these device manager views operate in a similar fashion.
Note: *For general device manager information, see the "About the Device Manager" section in the Drivers Guide.*
- Step 2 Click the **New** button to bring up the **New** device dialog.
Depending on the network type, the appropriate device object **Type** will be preselected (either ModbusAsyncDevice, ModbusTcpDevice, or ModbusTcpGatewayDevice).

For more details on this step, see one of the following:

- “[Modbus Async Device Manager notes](#)” on page 4-3
- “[Modbus Tcp Device Manager notes](#)” on page 4-5
- “[Modbus Tcp Gateway Device Manager notes](#)” on page 4-8

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

This brings up a dialog to name the device(s), enter Modbus device address, as well as enter other information, such as Modbus Config overrides, and ping address. Typically, the default values for most items are sufficient to start with, except the following:

- Any `ModbusAsyncDevice` needs the unique Modbus address (1–247) in use.
- Any `ModbusTcpDevice` needs the unique IP address in use (Modbus address can remain at 1). Also, if it uses a TCP port besides the standard 502, enter it here.
- Any `ModbusTcpGatewayDevice` needs the unique Modbus address (1–247) in use.

For more details on this step, see “[About Modbus client devices](#)” on page 4-12.

Step 4 Click **OK** to add the client device(s) to the network.

You should see the device(s) listed in the Modbus device manager view, showing a status of “{ok}” and enabled as “true.”

If a device shows “down” check the configuration of the network and/or the device addresses. You can simply double-click a device in the device manager to review settings in an **Edit** dialog, identical to the **New** dialog when you added it.

After making any address changes, click **Save**, then right-click the device and select **Actions > Ping**.

Note: *The default “ping address” for a client Modbus device is for the “first” (40001 Modbus) holding register value (integer)—often this 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. See “[Ping Address properties](#)” on page 4-13 for more details.*

Create Modbus client 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 the Point Manager (and in this case, the “**Modbus Client Point Manager**”). You use it to add Modbus client proxy points under any client Modbus device.


For general information, see the “About the Point Manager” section in the *Drivers Guide*. Also see “[Modbus Client Point Manager notes](#)” on page 4-16.

Note: *Unlike the point managers in many other drivers, the **Modbus Client Point Manager** does not offer a “Learn mode” with a **Discover** button and pane. The simplicity of the Modbus protocol excludes these functions. Instead, you simply use the **New** button to create proxy points, referring to the vendor’s documentation for the addresses of data items in each Modbus device.*

To add Modbus client proxy points

Once a client Modbus device is added, you can add proxy points to read and write data. 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. Use the following procedure:

To create client Modbus proxy points in a device:

Step 1 In the **Device Manager**, in the **Exts** column, double-click the **Points** icon  in the row representing the device you wish to create proxy points.

This brings up the **Modbus Client Point Manager**.

Step 2 (Optional) Click the **New Folder** button to create a new points folder to help organize points, and give it a short name, such as “Hldg1_to_64”, or whatever name works for your application. You can repeat this to make multiple points folders, or simply skip this step to make all proxy points in the root of **Points**. Note that all points folders have their own **Modbus Client Point Manager** view, just like **Points**. If making points folders, double-click one to move to its location (and see the point manager).

Step 3 At the location needed (**Points** root, or a points folder), click the **New** button.

The **New** points dialog appears, in which you select a point “Type,” “Number to Add,” “Starting Address,” and “Data Type” (latter applies only if selecting type: Numeric Point or Numeric Writable).

For more details, see “[About Modbus client proxy points](#)” on page 4-17.

- Step 4 Click **OK**.
- This brings up another **New** dialog to name the point(s), enter data addresses as well as enter other information, such as point facets and conversion. Default point names use a convention similar to: “<PointType><Address>”, for example: “Numeric Point40012” or “Boolean Writable6”.
- Details on related entries in this step are in “[Modbus client point ProxyExt properties](#)” on page 4-19, and background information is given in the section “[Modbus Meets NiagaraAX](#)” on page 3-1.
- Step 5 Click **OK** to add 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 have a status of “{ok}” with a polled value displayed.
- If a point shows 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”. In such a case, re-check the address in the point against the documented address for the data item. For related details, see “[Exception codes](#)” on page 3-10, and sections “[Data address format in NiagaraAX](#)” on page 3-5, and “[Data address format in NiagaraAX](#)” on page 3-5.
 - If a NumericPoint or NumericWritable for a float or long (2-register) value shows a “0” value (or an impossibly large value) instead of an expected value, yet still has an “{ok}” status, verify that the correct “byte order” settings exist for float and long values in the parent device. See “[Modbus Config \(client device level\)](#)” on page 4-12 for related details.
- Step 6 Continue to add proxy points as needed under the **Points** extension of each client Modbus device. As needed, double-click one or more existing points for the **Edit** dialog, similar to the **New** dialog used to create the points. This is commonly done for re-editing items like data addresses, names, or facets.
- Step 7 After all needed proxy points have been added under a device, you may wish to configure it for “device polling.” Typically, this improves polling response due to fewer messages to get the same amount of data. To configure a client Modbus device for device polling:
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/slots).
 2. Right-click **Device Poll Config**, and select **Actions > Learn Optimum Device Poll Config**.
- The necessary Device Poll Config Entry components are automatically added and configured under this container. For more details, see “[Configuring Device Poll Config](#)” on page 4-15.

Add other client Modbus components

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 following procedures explains how:

- [To add client Modbus presets](#)
- [To add client Modbus file records](#)

To add client Modbus presets

To add preset components under a client Modbus device:

- Step 1 Open the modbusAsync or 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 From the palette, drag the “Presets” 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** dialog, accept the default “Presets” 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.
- For more details, see “[About Modbus preset components](#)” on page 4-21.

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 if you copy these components under the Points container, they are not visible in any **Modbus Client Point Manager** view.

To add client Modbus file records

To add components for client Modbus file records, use the following procedure:

- 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** dialog, 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.
For more details, see “[About Modbus \(client\) file records](#)” on page 4-25.

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.

Configure the slave Modbus network

To add and configure a server (slave) Modbus network, perform the following main tasks:

- Add the server (slave) Modbus network, as needed:
 - [Add a ModbusSlaveNetwork](#)
 - [Add a ModbusTcpSlaveNetwork](#)
- [Review network-level Modbus client configuration](#)

Add a ModbusSlaveNetwork

Note: First see “[Modbus Driver Installations](#)” on page 1-1 for license and software requirements. For background information, see “[About Modbus Slave networks](#)” on page 4-8.

To add a ModbusSlaveNetwork in the station

Use the following procedure to add a `ModbusSlaveNetwork` component under the station’s Drivers container.

Note: If the host JACE has multiple RS-485 or RS-232 ports to be used for server (slave) Modbus networks, add one `ModbusSlaveNetwork` for each physical port. Note that for each `ModbusSlaveNetwork`, you must [Configure the serial port parameters and transmission mode](#).

To add a `ModbusSlaveNetwork` in the station:

- Step 1 Double-click the station’s **Drivers** container, to bring up the **Driver Manager**.
- Step 2 Click the **New** button to bring up the New network dialog. For more details, see “Driver Manager New and Edit” in the *Drivers Guide*.
- Step 3 Select “Modbus Slave Network,” number to add: 1 (or more if multiple networks) and click **OK**.
This brings up a dialog to name the network(s).
- Step 4 Click **OK** to add the `ModbusSlaveNetwork`(s) to the station.

You should have a `ModbusSlaveNetwork` named “ModbusSlaveNetwork” (or whatever you named it), under your Drivers folder, initially showing a status of “{fault}” and enabled as “true.”

After you [Configure the serial port parameters and transmission mode](#), status should change to “{ok}”.

Configure the serial port parameters and transmission mode

In the `ModbusSlaveNetwork` property sheet for each network, you must set the serial port configuration to match the serial communications parameters used to communicate to the attached Modbus master device, including the Modbus transmission mode (RTU or ASCII).

To set the serial port parameters

To set the serial port parameters and mode for a ModbusSlaveNetwork:

- Step 1 Right-click the ModbusSlaveNetwork and select **Views > Property Sheet**.
The **Property Sheet** appears.
- Step 2 Scroll down and expand the **Serial Port Config** slot
Set the properties for the JACE serial port used, where defaults are:
- Port Name: none — Enter the JACE port being used, like COM2 or COM3.
 - Baud Rate: Baud9600 — Or choose different from selection list.
 - Data Bits: Data Bits8 — Or choose different from selection list.
 - Stop Bits: Stop Bit1 — Or choose different from selection list.
 - Parity: none — Or choose different from selection list.
 - Flow Control Mode: none — Or choose different using checkbox.
- Note:** You must determine the setup of the Modbus serial network to correctly set the baud rate, data bits, stop bits, parity, and flow control settings.
- 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, you should review its "global" Modbus settings.
See "[Review network-level Modbus server configuration](#)" on page 2-7.

Add a ModbusTcpSlaveNetwork

Note: First see "[Modbus Driver Installations](#)" on page 1-1 for license and software requirements. For background information, see "[About Modbus TCP Slave networks](#)" on page 4-10.

To add a ModbusTcpSlaveNetwork in the station

Use the following procedure to add a ModbusTcpSlaveNetwork component under the station's Drivers container.

Note: Only one ModbusTcpSlaveNetwork is needed, even if the JACE host has two Ethernet ports connected to two different (non-routed) TCP/IP LANs. In this case, the destination IP addresses of child ModbusTcpSlaveDevices will automatically determine the Ethernet port utilized.

To add a ModbusTcpSlaveNetwork in the station:

- Step 1 Double-click the station's **Drivers** container, to bring up the **Driver Manager**.
- Step 2 Click the **New** button to bring up the New network dialog. For more details, see "Driver Manager New and Edit" in the *Drivers Guide*.
- Step 3 Select "Modbus Tcp Slave Network," number to add: 1 and click **OK**.
This brings up a dialog to name the network.
- Step 4 Click **OK** to add the ModbusTcpSlaveNetwork to the station.
You should have a ModbusTcpSlaveNetwork named "ModbusTcpSlaveNetwork" (or whatever you named it), under your Drivers folder, initially showing a status of "{ok}" and enabled as "true."
- Step 5 Right-click the ModbusTcpSlaveNetwork and select **Views > Property Sheet**.
The **Property Sheet** appears. See the next section, "[Review network-level Modbus server configuration](#)".

Review network-level Modbus server configuration

For either of the server Modbus networks (ModbusSlaveNetwork, ModbusTcpSlaveNetwork), you should review its "network-level" defaults for interpreting "2-register" numerical data (floats and longs). These defaults are on the property sheet of the network-level component.

Note: These settings apply to all Modbus slave devices in the network, unless overridden in any device's configuration, using equivalent properties. For details, see "[Modbus Config \(server device level\)](#)" on page 4-26.

To review network-level server Modbus configuration

To review network-level Modbus configuration:

- Step 1 In the property sheet of the network, review the following properties:
- Float Byte Order: Order3210 — Or select Order1032 instead if used by most devices.
 - Long Byte Order: Order3210 — Or select Order1032 instead if used by most devices.
- Step 2 If you made any changes, click the **Save** button.

Add server (slave) Modbus devices

After adding a slave Modbus network, you can use the network's default “device manager” view to add the appropriate server (slave) Modbus devices.

Note: *You need to plan how you are mapping station data into one or more “virtual” Modbus slave devices, including the “virtual” Modbus data items like coils, inputs, input registers, and holding registers.*
In the case of a ModbusSlaveNetwork, you may wish to create multiple “virtual” Modbus slave devices, each with a different (and currently unused) Modbus address. Or, you may simply wish to have all data to be exposed to a Modbus master to appear sourced from a single (one address) device.
In the case of ModbusTcpSlaveNetwork, although you can add multiple “virtual” Modbus slave devices (with different Modbus addresses)—they all must be reached through the same IP address as the station.
In any case, for any Modbus slave device, you can configure a number of valid data item “register ranges” for it. In turn, you can then add corresponding Modbus server proxy points under that device, in order to exchange data with the networked Modbus server.
Essentially, you are building a “custom” Modbus device, and should document its setup (and application) carefully, to make available to the programmer of the master Modbus device.

The following two procedures describe needed procedures:

- [To add a server \(slave\) Modbus device in the network](#)
- [To configure register ranges in a Modbus slave device](#)

To add a server (slave) Modbus device in the network

Use the following procedure to add the correct type of slave Modbus device in the network.

To add a sever (slave) Modbus device:

- Step 1 In the Nav tree or in the Driver Manager view, double-click the slave network, to bring up the device manager (Modbus Slave Device Manager, Modbus Tcp Slave Device Manager). Both of these device manager views operate in a similar fashion.
- Note:** *For general device manager information, see the “About the Device Manager” section in the Drivers Guide.*
- Step 2 Click the **New** button to bring up the **New** device dialog.
Depending on the network type, the appropriate device object **Type** will be preselected (either ModbusSlaveDevice or ModbusTcpSlaveDevice).
- Step 3 Select for number to add: 1 (or more, if multiple) and click **OK**.
This brings up a dialog to name the device(s), enter Modbus device address, and set Modbus Config overrides. Typically, the default values for most items are sufficient to start with, except the following:
- Any ModbusSlaveDevice needs a unique Modbus address (1—247), not currently in use.
For more details on this step, see [“About Modbus server devices”](#) on page 4-26.
- Step 4 Click **OK** to add the server device(s) to the network.
You should see the device(s) listed in the Modbus device manager view, showing a status of “{ok}” and enabled as “true.”

To configure register ranges in a Modbus slave device

Use the following procedure to establish the “register ranges” for data items in any server (slave) Modbus device. The device's proxy points must fall within these register ranges, or else they will have a fault status.

To configure register ranges in a server (slave) Modbus device:

- Step 1 In the Nav tree, expand the slave network, and double-click the slave Modbus device of interest.
Its property sheet show, including 4 “Modbus Register Range Table” container slots with default names “Valid Coils Range,” “Valid Status Range,” “Valid Input Register Range,” and “Valid Holding Register Range”.
- Note:** *Alternatively, you can access these container slots when you expand the Modbus slave device in the Nav tree, if you prefer to work from the Nav side bar.*
- Step 2 Click to expand each of these range containers, and expand the single “Default” register range entry in each. By default, a slave device copied from the modbusSlave or modbusTcp palette has the same values for each of the 4 default register range containers: Enabled, Starting Address Offset: 1, Size: 64.
- Step 3 Make whatever range entry changes are needed, and click **OK**.
Note that you can add *additional* register range entries using the right-click “Add Range” action on any of the 4 register range containers.
For more details, see [“Modbus Register Range Tables”](#) on page 4-27.

Create Modbus server proxy points

As with device objects in other drivers, each server Modbus device has a **Points** extension that serves as the container for proxy points. The default view for any Points extension is the Point Manager (and in this case, the “**Modbus Server Point Manager**”). You use it to add Modbus server proxy points under any server (slave) Modbus device.

For general information, see the “About the Point Manager” section in the *Drivers Guide*.

Note: *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. See the previous procedure, “To configure register ranges in a Modbus slave device” on page 2-8.*

To add Modbus server proxy points

Once a slave Modbus device is added and its register ranges defined, you can add server proxy points. Use the following procedure:

To create server Modbus proxy points in a device:

- Step 1 In the **Device Manager**, in the **Exts** column, double-click the **Points** icon  in the row representing the device you wish to create proxy points.
This brings up the **Modbus Server Point Manager**.
- Step 2 (Optional) Click the **New Folder** button to create a new points folder to help organize points, and give it a short name, such as “Hldg1_to_64”, or whatever name works for your application. You can repeat this to make multiple points folders, or simply skip this step to make all proxy points in the root of **Points**. Note that all points folders have their own **Modbus Server Point Manager** view, just like **Points**. If making points folders, double-click one to move to its location (and see the point manager).
- Step 3 At the location needed (**Points** root, or a points folder), click the **New** button.
The **New** points dialog appears, in which you select a point “Type,” “Number to Add,” “Starting Address,” and “Data Type” (latter applies only if selecting type: Numeric Point or Numeric Writable).
Note: *Generally, it is unwise to expose any coil or holding register as a writable point if the Modbus master may also write to this same item—otherwise “write contention” issues may result. In other words, writable point types are better suited to items exposed as Modbus (status) “inputs” and “input register s”.*
For more details, see “[About Modbus server proxy points](#)” on page 4-29 and “[Types of Modbus server proxy points](#)” on page 4-29.
- Step 4 Click **OK**.
This brings up another **New** dialog to name the point(s), enter data addresses as well as enter other information, such as point facets and conversion. Default point names use a convention similar to: “<PointType><Address>”, for example: “Numeric Point40012” or “Boolean Writable6”.
Details on related entries in this step are in “[Modbus server point ProxyExt properties](#)” on page 4-30, and background information is given in the section “[Modbus Meets NiagaraAX](#)” on page 3-1.
- Step 5 Click **OK** to add 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 have a status of “{ok}” with a value displayed.
If a point shows 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”.
In such a case, re-check the address in the point against the defined register ranges in the parent slave device. For related details, see “[Exception codes](#)” on page 3-10, and sections “[Data address format in NiagaraAX](#)” on page 3-5, and “[Modbus Register Range Tables](#)” on page 4-27.
- Step 6 Continue to add proxy points as needed under the **Points** extension of each server Modbus device.
As needed, double-click one or more existing points for the **Edit** dialog, similar to the **New** dialog used to create the points. This is commonly done for re-editing items like data addresses, names, or facets.

Add other server Modbus components

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.

To add server Modbus file records

To add components for server Modbus file records, use the following procedure:

- 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** dialog, 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. For more details, see [“About Modbus \(server\) file records”](#) on page 4-31.

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 3

Modbus Meets NiagaraAX

This section provides a quick look at Modbus. It is by no means a complete summary or overview. Only key points necessary for successful NiagaraAX integrations are included. A good understanding of these concepts will make integration easier.

The following main sections are included:

- [Brief History](#)
- [Modbus key concepts](#)
 - [Modbus registers](#)
 - [Modbus data addresses](#)
 - [Modbus data types](#)
 - [Modbus function codes](#)
 - [Modbus messages](#)
 - [Exception responses](#)
- [Device-to-device differences](#)

Note: *Currently, the independent, member-based, non-profit organization **Modbus-IDA** provides a variety of links to Modbus technical resources. Included are technical overviews, FAQs, and complete protocol (standards) documents. For detailed Modbus information, refer to it at this URL: www.modbus.org*

Brief History

Modbus is an open communications protocol originally developed in 1978 by Modicon Inc.¹ for networking industrial PLCs (programmable logic controllers). 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.

Apart from PLCs, Modbus-capable devices now include many with both industrial and commercial applications, such as electric-demand meters and lighting controllers, among many others. In addition, Modicon has introduced another variant of the Modbus protocol, Modbus TCP, again as an open protocol. Modbus TCP 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. Modbus Plus is currently not supported by any NiagaraAX driver.*

Modbus key concepts

The Modbus protocol defines a message structure and format used in communication transactions. Modbus devices communicate using a master-slave method, in which only the master device can initiate a communications transaction. There can be only **one** master device on a Modbus network—in most integrations (modbusAsync, modbusTcp), this is the JACE. All other devices must be Modbus slaves.

However, note that two “slave” Modbus drivers are also available, in which the station can act as a dumb slave (server), using modbusSlave or modbusTcpSlave components. Usage of these drivers is expected to be infrequent. However, basic Modbus principles remain the same.

1. Modicon is now an international brand of Schneider Electric.

Before reading about Modbus, please understand that any NiagaraAX Modbus integration uses Modbus proxy points to provide monitoring and control, similar to other NiagaraAX integrations. Therefore, many of the following Modbus topics mention items specific to NiagaraAX components, to help clarify station configuration.

The following topics may contribute to a basic understanding of Modbus:

- [Modbus registers](#)
- [Modbus data addresses](#)
- [Modbus data types](#)
- [Modbus function codes](#)
- [Modbus messages](#)
- [Exception responses](#)

Modbus registers

A Modbus device holds transient (real-time) data and often persistent (configuration) data in addressable **registers**. Here, the term “registers” implies all addressable data, but this is a loose interpretation. Using Modbus nomenclature, all accessible data in a Modbus slave is contained in the following four available groups of data flags and registers (including the Modbus-master access that is possible):

- **Coil status**
Or simply “*coils*”: single-bit flags that represent the status of digital (NiagaraAX: boolean) outputs of the slave, that is, On/Off output status. A Modbus master can both **read** from **and write** to coils.
- **Input status**
Or simply “*inputs*”: single-bit flags that represent the status of digital (NiagaraAX: boolean) inputs of the slave, that is, On/Off output status. A Modbus master can **read (only)** inputs.
- **Input registers**
Are 16-bit registers that store data collected from the field by the Modbus slave. The Modbus master can **read (only)** input registers.
- **Holding registers**
Are 16-bit registers that store general-purpose data in the Modbus slave. The Modbus master can both **read** from **and write** to input registers.

Modbus 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.

The following sections provide more details:

- [Organization and addressing of data](#)
- [Consecutive addresses](#)
- [Consecutive address usage \(general\)](#)
- [Consecutive address usage \(NiagaraAX\)](#)
- [Data address format in NiagaraAX](#)

Organization and addressing of data

Modbus data in a device is addressed as follows:

- **Coils** — Addressed at 00000 — 0nnnn decimal, or “0x” addresses.
- **Inputs** — Addressed at 10000 — 1nnnn decimal, or “1x” addresses.
- **Input Registers** — Addressed at 30000 — 3nnnn decimal, or “3x” addresses.
- **Holding Registers** — Addressed at 40000 — 4nnnn decimal, or “4x” addresses.

Note that 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, as shown for a meter in [Table 3-1](#).

Table 3-1 Example Modbus 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	kW	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)
—	—	—	—
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

In the [Table 3-1](#) example, each table row represents a 16-bit holding register. The holding register with **Modbus address** 40011 (phase B power), can be alternately addressed as a holding register (with an “implied 4nnnn value”) having a **decimal address** of “10” (“0010”), or a **hex address** of “A” (“000A”).

Note: Integer-stored data often requires additional math operations. NiagaraAX Modbus proxy points provide this capability “built-in,” via the proxy extension’s **Conversion** slot. For example, selecting “Linear” in Conversion provides entries for scale and offset, to process the raw Modbus value to a finished value. Typically, the vendor’s documentation for a Modbus device includes any “scaling” values needed for any such data items.

Consecutive addresses

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 [Table 3-1](#) example, holding registers 40001-40015 are used consecutively for integer data, where each is a separate, integer, “data point”.

In this example device, register 40259 begins a consecutive series of holding registers used to access “floating point” data values—note, however, that an address “gap” exists in this case. The address gap (while not necessary), was probably implemented by the device vendor for clarity. Note also that floating-point data values (being 32-bit based) require the use of **two consecutive registers** for each data point. See “Numerical data types” on page 3-6 for related information.

Consecutive address usage (general)

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.

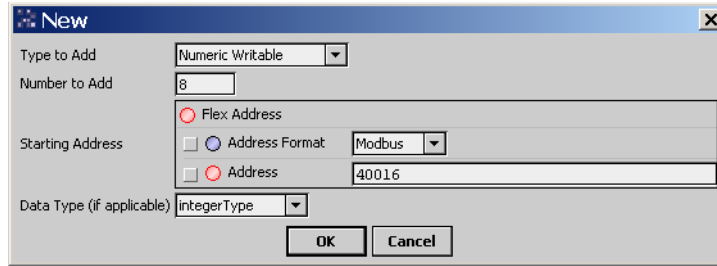
The address range for data in any data group (coils, inputs, input registers, holding registers) received in a query must be implemented by the receiving device—otherwise, it will generate an exception response. For example, a read request of holding registers 40003—40015 to the device represented by [Table 3-1](#) 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). See Exception Responses for more information.

Consecutive address usage (NiagaraAX)

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

- When using the Point Manager to create proxy points, by using the “Number to Add” option in the **New** dialog box. See [Figure 3-1](#) below for an example of where 8 points are being added.

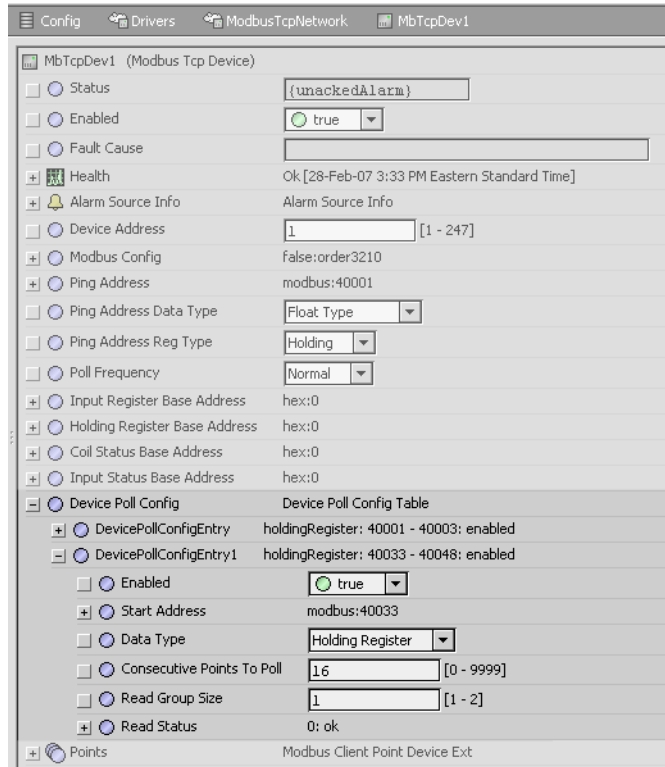
Figure 3-1 New dialog to create Modbus proxy points has “Number to Add” option for consecutive points



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.

- 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, reducing network messaging traffic. In NiagaraAX, this is configured in the *device* object (ModbusAsyncDevice, ModbusTcpDevice, ModbusTcpGatewayDevice), in the device’s DevicePollConfigTable slot. See [Figure 3-2](#) below for an example.

Figure 3-2 Modbus client device’s DevicePollConfigTable container slot allows “device polling” setup

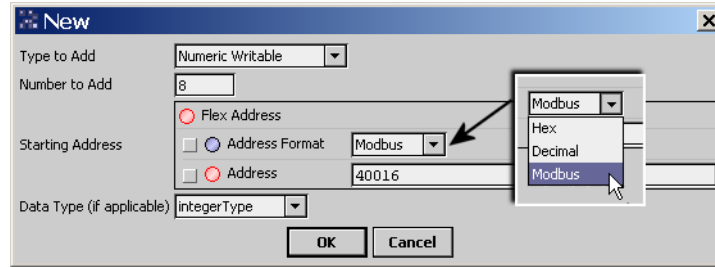


You can add child “DevicePollConfigEntry” objects manually in this container and configure, or optionally use the container’s right-click *action*: “Learn Optimum Device Poll Config”. Note that device polling should be configured only after proxy points are created, and typically already receiving values from (individual) point polling. For more details, see “[Device Poll Config](#)” on page 4-15.

Data address format in NiagaraAX

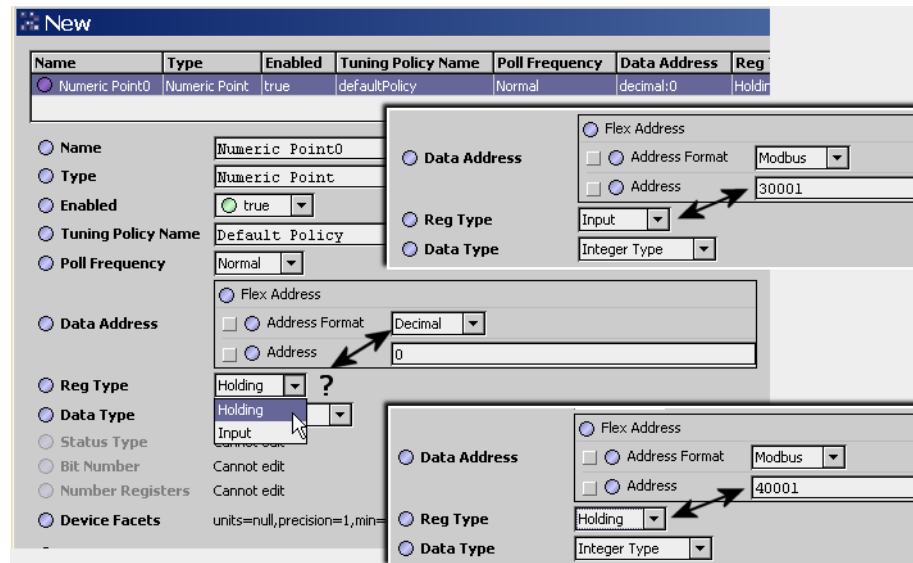
Generally speaking, when configuring a Niagara Modbus point for a data address, you should choose “Modbus” addressing from the available “Address Format” types (Figure 3-3). This lets you enter target data addresses directly from the device’s documentation, without having to “subtract 40001” for example, or perform other mental math.

Figure 3-3 Modbus address format recommended when adding proxy points



Also, for read-only client points, using the Modbus address format also frees you from setting the register type property (“Reg Type”), as it automatically sets this property going by the leading numeral of the full Modbus address (3 for input registers, and 4 for holding registers), as shown in Figure 3-4.

Figure 3-4 Reg Type property of Modbus



Note: When entering a Modbus formatted address for a coil, “leading zeros” are ignored—for example the Modbus address “00109” is the same as entering Modbus “109”. Again, note that 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.

Modbus data types

Data in Modbus devices can exist in a wide variety of ways. In a NiagaraAX Modbus integration, data is brought into Niagara's common object model in a manner that simplifies the sharing of values. See the following subsections for details:

- [Data in Modbus](#)
- [Numerical data types](#)
- [NiagaraAX data representation](#)

Data in Modbus

By definition, data represented by *coils* and *inputs* is “status”, which in Modbus nomenclature means “boolean” (On/Off). In Niagara, this equates to two-state, meaning Active or Inactive (true or false).

As far as *input registers* and *holding registers* are concerned, however, the Modbus protocol does not dictate how data is formatted (data type used, numerical encoding). A Modbus device vendor can use whatever data types are needed and can be accomplished with one or more consecutive 16-bit registers.

Note: *The device vendor should document the data type used for each input register and holding register—and it is important to configure Niagara Modbus proxy points accordingly. When dealing with 32-bit values such as long or float values (see [Numerical data types](#) below), this includes the “byte-order scheme” for the two consecutive registers, as processed in the device.*

Perhaps the most popular data type for a register is “integer,” using a single register. This is the Niagara “Data Type” for an unsigned, 16-bit integer value, and is the *default* data type for many newly-created Modbus proxy points. The value range possible in an integer value is 0 to 65,535. Note that in some systems (or devices), the term *word* is synonymous, that is, meaning an unsigned, 16-bit integer value.

Numerical data types

The following numerical data types are supported by Niagara proxy points for Modbus input registers and holding registers:

- **Integer** — Unsigned 16-bit integer, data range 0 to 65,535. Same as “word”.
- **Float**— (floating point), 32-bit single precision, sometimes called “real”. Very small and large numbers are possible. Requires two consecutive registers¹ in the Modbus device. In addition, there are two different byte-order schemes for float values (3-2-1-0 or 1-0-3-2).
- **Long** — Signed 32-bit integer, data range -2,147,483,648 to 2,147,483,647. Also requires two consecutive registers, and the same byte-order scheme information (as for “float” data).
- **Signed Integer** — Signed 16-bit integer, data range -32,768 to 32,767. Sometimes called “short”.

For any Modbus proxy NumericPoint or NumericWritable, you specify this in the “Data Type” property of its Modbus proxy extension.

Note that the “byte order” scheme for two-register numeric values (float and long) can be set at the device level (if necessary), or set globally at the network level. See “[Device-to-device differences](#)” on page 3-11 for related details.

In addition to these numeric formats, sometimes a holding register or input register is used by a device to pack a number of *boolean* statuses (On/Off states), with *each status* mapped to *a bit*. NiagaraAX Modbus provides special “Register Bit” proxy point extensions for boolean control points to read and write to such registers, accessing each bit independently per proxy point.

Also (although not common), a Modbus device may use a number of consecutive holding registers to hold a string of alphanumeric (ASCII encoded) characters. The Modbus integration provides a “Modbus String” proxy point extension for String control points to read such character strings.

1. Proxy points automatically allocate 2 consecutive registers for each data item whenever you specify a float or long data type. Keep this in mind when specifying “number of points” within any range of Modbus registers.

NiagaraAX data representation

With the exception of the String control point with the “Modbus String” proxy point extension, Modbus proxy points represent all data on their inputs and outputs using these Niagara data types:

- **Boolean** — BooleanPoint and BooleanWritable. Two-state data represented by coils and inputs. Less frequently, boolean data is “bitmapped” into input registers or holding registers. Note that all Modbus boolean proxy points provide facets that you can individually edit to match the vendor’s documented state descriptions, such as “On” and “Off”, or “enable” and “disable”.
- **Numeric** — NumericPoint and NumericWritable. Numerical data in holding registers or input registers, whether a Modbus Float proxy point’s selected Data Type is integer, long, float, or signed integer (see [Numerical data types](#)). Note that all Modbus numeric proxy points provide facets that you can individually edit for minimum/maximum values, precision, and data units.
- **Enum** — (Modbus 3.5.26 or higher, or AX-3.6 or later) EnumPoint and EnumWritable. Enumerated data in holding registers or input registers, using the integer (ordinal) value resulting from some range of consecutive bits, specified by a starting bit and number of bits. See [“Example”](#) on page 4-18.

Rounding values Note that Modbus NumericWritable proxy points that write values to holding registers provide rounding (and possibly clamping) of the input value before any write. This depends on the proxy point’s selected Data Type, as follows:

- **Integer type** — Input values are rounded up or down to the nearest whole number. Range is from 0 to 65,535; input values outside the range are clamped to these limits.
- **Long type** — Input values are rounded up or down to the nearest whole number. The range (-2,147,483,648 to 2,147,483,647) matches the Niagara value range.
- **Signed Integer type** — Input values are rounded up or down to the nearest whole number. Range is from -32,768 to 32,767; input values outside this range are clamped to these limits.
- **Float type** — No rounding—any input value is written directly as is.

Modbus function codes

Modbus uses defined function codes in communications transactions for the type of information requested by the master, in addition to a specific data addresses (or range of addresses) that apply to the function code-defined request. 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 different function codes. However, few devices support all function codes. A vendor’s documentation for a Modbus device should state which function codes are supported.

Supported function codes

Function codes have associated numbers, used in Modbus messages. [Table 3-2](#) shows in numerical order the function codes supported through the use of components in any NiagaraAX Modbus integration.

Table 3-2 NiagaraAX-supported Modbus function codes

Code	Function Name	NiagaraAX Operation (JACE master)
01	READ COIL STATUS	Normal data polling of all read-only and writable Modbus proxy points.
02	READ INPUT STATUS	
03	READ HOLDING REGISTERS	
04	READ INPUT REGISTERS	
05	FORCE SINGLE COIL	Change of values at inputs and/or invoked actions of writable Modbus proxy points or client “preset” components.
06	PRESET SINGLE REGISTER	
07	READ EXCEPTION STATUS ^a	Poll device for exception status, output values and bits set.
15	FORCE MULTIPLE COILS	Change of values at inputs and/or invoked actions of writable Modbus proxy points or client “preset” components.
16	PRESET MULTIPLE REGISTERS	
20	READ FILE RECORD	Using ModbusClientStringRecord to read and write file records, in which data is converted to ASCII characters and displayed as string.
21	WRITE FILE RECORD	

a. Modbus AX-3.5 or later (build 3.4.52 or higher) required. See [“About Modbus client exception status”](#) on page 4-25.

Note: *If integrating a Modbus device that supports functions codes 15 or 16 (see [Table 3-2](#)), you can optionally choose to use these function codes instead of codes 05 and 06 by setting the corresponding Modbus Config properties (in the Modbus network and/or device objects) from “false” value defaults. See [“Device-to-device differences”](#) on page 3-11 for related details.*

Modbus messages

Each communications transaction is *initiated* by the Modbus *master*, with a message known as a Query. To any specifically addressed query, the master expects a Response back from the slave. A slave never initiates a transaction (sends an unsolicited message). This query-response cycle is the basis for all communications on a Modbus network. It is always the master that initiates the query, and the slave that responds.

A “broadcast” message can also be sent by the Modbus master. In this one case, the master does not require a response from any slave. A slave identifies a received broadcast message by the target address of “device 0”. See the following subsections for more details:

- [Message structures](#)
- [NiagaraAX debug example](#)

Message structures

Each Modbus message has a defined structure, referencing device address, function code, and the data address (or requested data). Message structures in a serial (async) network are as follows:

Query The master-generated query message contains, in order:

1. A *device address* of target slave (or “broadcast address”, for all slaves), as first byte.
2. A *function code* defining the requested action, as second byte.
3. A *data* field describing particulars for the function code. This may be a register address (and a range) to read, or a coil address to write, for example.
4. An *error-check* field to confirm integrity of the message, as it will be received from the master. If the slave detects an error in a query, it is ignored. The slave then waits for the next query addressed to it.

Response The slave-generated response message contains, in order:

1. Its *device address*, confirming to the master that it is replying to the query, as first byte.
2. The *function code*, as second byte—normally an exact copy, unless the slave is unable to perform the requested function. In this case, the function code returned is a modified form, indicating that the slave was unable to perform. See “[Exception responses](#)” on page 3-9 for more details.
3. The *data*, containing the data requested in the query.
4. An *error-check* field to confirm integrity of the message, as it will be received from the slave. If the master detects an error in the response, it is ignored.

The error-checking method depends on the Modbus transmission mode—in Modbus RTU (the most prevalent), a CRC (cyclical redundancy checksum) method is used.¹

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

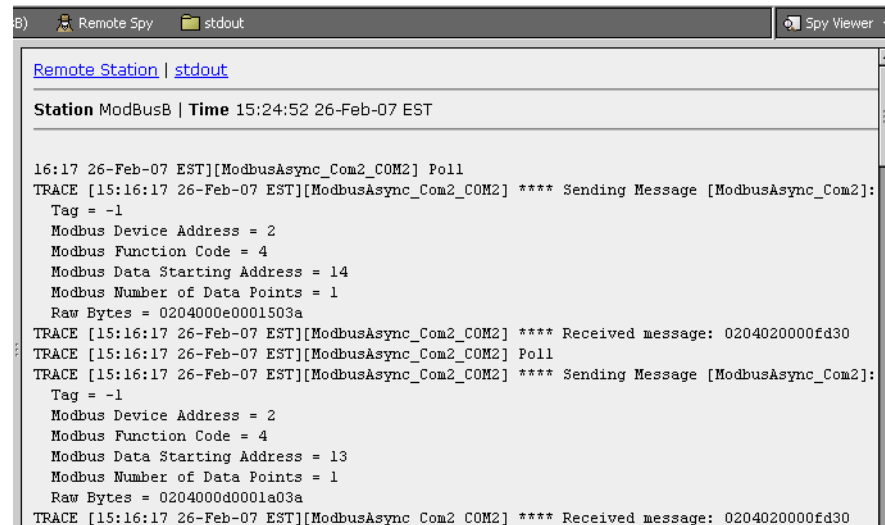
- **Query**
020300030004B43A
For device 02, function code 03, starting address 0003, number of registers 04, error checksum B43A
- **Response**
02030800510052003C003CA387
From device 02, function code 03, number bytes returned 08, data (00510052003C003C), error checksum A387

1. The Modbus TCP protocol has a similar message format for query and response messages. However, Modbus TCP is freed from error check routines. Instead, the error-checking mechanisms built into the lower-level TCP/IP and link layers (that is, Ethernet) are used.

NiagaraAX debug example

By enabling “Trace” logging on a Modbus network, you can see this query/response message cycle in the station’s Standard Output, for example resulting from normal data polling. The sent query is “broken out” to show fields on separate lines, and the received (response) is shown in a single line (in hex format).

Figure 3-5 Trace-level log output from (COMx) of a ModbusAsyncNetwork, as seen in Standard Output



```
Remote Station | stdout
Station ModBusB | Time 15:24:52 26-Feb-07 EST

16:17 26-Feb-07 EST][ModbusAsync_Com2_COM2] Poll
TRACE [15:16:17 26-Feb-07 EST][ModbusAsync_Com2_COM2] **** Sending Message [ModbusAsync_Com2]:
  Tag = -1
  Modbus Device Address = 2
  Modbus Function Code = 4
  Modbus Data Starting Address = 14
  Modbus Number of Data Points = 1
  Raw Bytes = 0204000e0001503a
TRACE [15:16:17 26-Feb-07 EST][ModbusAsync_Com2_COM2] **** Received message: 0204020000fd30
TRACE [15:16:17 26-Feb-07 EST][ModbusAsync_Com2_COM2] Poll
TRACE [15:16:17 26-Feb-07 EST][ModbusAsync_Com2_COM2] **** Sending Message [ModbusAsync_Com2]:
  Tag = -1
  Modbus Device Address = 2
  Modbus Function Code = 4
  Modbus Data Starting Address = 13
  Modbus Number of Data Points = 1
  Raw Bytes = 0204000d0001a03a
TRACE [15:16:17 26-Feb-07 EST][ModbusAsync_Com2_COM2] **** Received message: 0204020000fd30
```

Note that 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. There, a 6-byte leading TCP header “000000000006” is seen in sent queries, and the checksum byte is omitted in both sent and response messages.

Exception responses

If a Modbus slave receives a query message correctly (that is, it passes error checking), but then finds it is unable to perform the required operation, it issues 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:

Exception response format

A slave-generated response message contains, in order:

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. See “Exception codes”.
4. An **error-check** field to confirm integrity of the message, as it will be received from the slave. If the master detects an error in the response, it is ignored.

Exception codes

Table 3-3 lists standard Modbus exception codes (01-08) plus extended codes (09-13). NiagaraAX 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, as in the table below.

Table 3-3 *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.
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 Niagara 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.

Device-to-device differences

Because Modbus does not specify *which* specific function codes are necessary in a device (much less the data group types needed), devices from different vendors tend to greatly vary. In addition, the data format of register-held data, as previously mentioned, is left up to the vendor. Quite commonly, different byte-order storage schemes are used for storing 32-bit data types such as float and long (integer).

With this in mind, the 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* (see “Device-level Modbus Config properties”). Properties are:

- [Network-level settings](#)
- [Device-level Modbus Config properties](#)

Network-level settings

The four properties, as they appear in a Modbus network include two for the “byte-order” that devices use for sending or receiving 4-byte (32-bit) data values in Modbus messages. Separate settings are for float (floating-point) and long (integer).

- **Float Byte Order**
Choices reflect two alternate methods, where numerals 0, 1, 2, and 3 represent the least significant byte to most significant byte, as follows:
 - `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”.
- **Long Byte Order**
Choices reflect two alternate methods, where numerals 0, 1, 2, and 3 represent the least significant byte to most significant byte, as follows:
 - `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”.

Note: Float or long values received in incorrect byte order may appear abnormally big, or not at all.

The other two properties depend on the Modbus function codes supported by child devices, which (if available) provide alternative options in Modbus messaging for write actions:

- **Use Preset Multiple Register**
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 used in place.
- **Use Preset Multiple Coil**
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 used in place.

Device-level Modbus Config properties

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, as follows:

- **Override Network**
True or false. The default, `false`, means the [Network-level settings](#) are used. Set this to True whenever you want the settings below to be used instead of the equivalent network-level values.
- **Float Byte Order**
Choices reflect two alternate methods, where numerals 0, 1, 2, and 3 represent the least significant byte to most significant byte, as follows:
 - `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”.
- **Long Byte Order**
Choices reflect two alternate methods, where numerals 0, 1, 2, and 3 represent the least significant byte to most significant byte, as follows:
 - `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”.
- **Use Preset Multiple Register**
Function code 16 support (Preset Multiple Registers) is available in the device (true or false). The default is *false*, where function code “Preset Single Register” is used in place.
- **Use Preset Multiple Coil**
Function code 15 support (Preset Multiple Coils) is available in the device (true or false). The default is *false*, where function code “Preset Single Coil” is used in place.

CHAPTER 4

NiagaraAX Modbus Representation

There are four different types of Modbus drivers (and five network types). Three networks are *client* types, where the station acts as a Modbus master device. The other two networks are slave or *server* types, where the station exposes data as a Modbus server, and simply responds to Modbus queries.

All Modbus networks use the standard NiagaraAX network architecture. See “About Network architecture” in the *Drivers Guide* for more details. These are the different Modbus network types:

- Client types:
 - Modbus Async — see [About Modbus Async networks](#).
 - Modbus Tcp — see [About Modbus TCP networks](#).
 - Modbus Tcp Gateway — see [About Modbus TCP Gateway networks](#).
- Server types:
 - Modbus Slave — see [About Modbus Slave networks](#).
 - Modbus Tcp Slave — see [About Modbus TCP Slave networks](#).

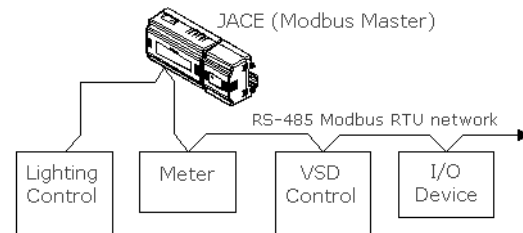
Under the various Modbus networks, child Modbus device components and their children (proxy points and other components) are similar and sometimes identical—especially among the client types, and also among server types. Therefore, Modbus component information is arranged into two main sections:

- [About Master \(client\) types](#)
- [About Slave \(server\) types](#)

About Modbus Async networks

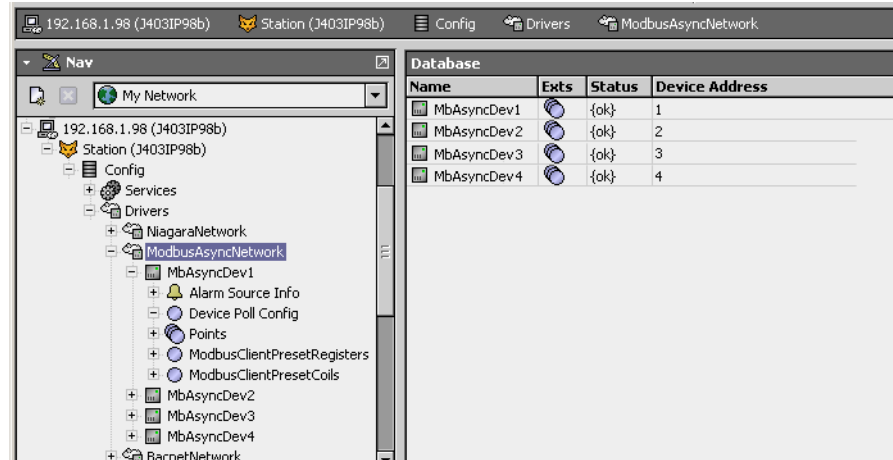
Each ModbusAsyncNetwork requires a serial port (typically, RS-485) on the host JACE platform. The JACE connects on the Modbus RTU or ASCII network and functions as the Modbus master.

Figure 4-1 ModbusAsyncNetwork with JACE as master device, here on an RS-485 Modbus RTU 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 JACE may support multiple Modbus Async networks.

Figure 4-2 ModbusAsyncNetwork in JACE station



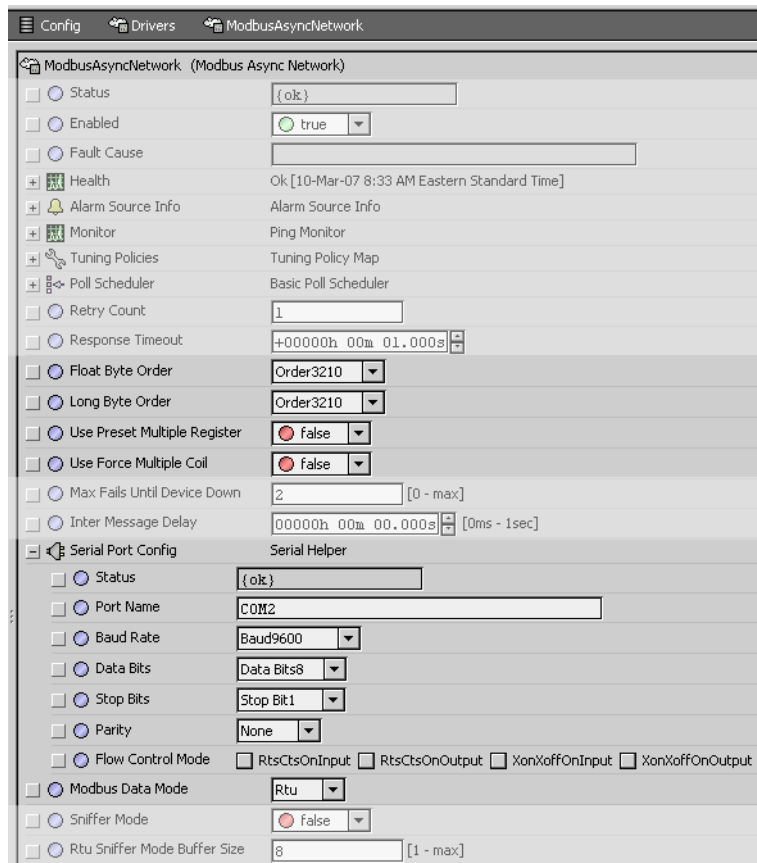
The station acts as Modbus master to all other Modbus devices on the attached COM port. Each child device is represented by a ModbusAsyncDevice, and has a unique Modbus address (1 to 247), as well as other Modbus config data and starting addresses for Modbus data items (coils, inputs, input registers,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

Modbus Async Network configuration is straightforward, using the property sheet of the network object.

Modbus Async Network configuration

In the property sheet of the ModbusAsyncNetwork (Figure 4-3), you configure the serial port/comm settings needed to communicate to attached Modbus devices, and review other default network values.

Figure 4-3 ModbusAsyncNetwork property sheet



Note the ModbusAsyncNetwork has the standard collection of network components, such as for status, health, monitor, tuning policies, and poll scheduler. For more details, see “Common network components” in the *Drivers Guide*.

In addition, the following properties have special importance:

- “Global” Modbus device defaults, which (as needed) can be overridden in the [Modbus Config \(client device level\)](#) slot of any child ModbusAsyncDevice:
 - **Float Byte Order**
Specifies the byte-order used to assemble or receive floating-point (32-bit) values in messages.
 - **Long Byte Order**
Specifies the byte-order used to assemble or receive long integer (32-bit) values in messages.
 - **Use Force Multiple Coil**
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.
 - **Use Preset Multiple Register**
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.
- **Serial Port Config**
The container slot in which you specify the serial port/communications setup required to talk to attached serial Modbus devices.
- **Modbus Data Mode**
As either RTU (the default) or ASCII, depending on the type of networked Modbus devices.

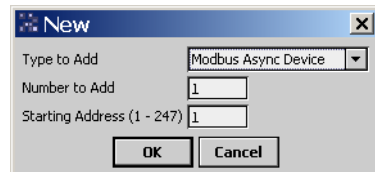
Other ModbusAsyncNetwork properties such as “Retry Count”, “Response Timeout”, “Sniffer Mode”, and “Rtu Sniffer Mode Buffer Size” are typically left at defaults, unless particular reasons dictate change.

Double-click the ModbusAsyncNetwork for the default **Modbus Async Device Manager** view, which you can use to add new ModbusAsyncDevice children. For general information, see “About the Device Manager” in the *Drivers Guide*. See “[Modbus Async Device Manager notes](#)” for additional details.

Modbus Async Device Manager notes

Each ModbusAsyncDevice you add under a ModbusAsyncNetwork requires the unique Modbus address (1—247) used by that device on that network. When using the Modbus Async Device Manager view to create new devices, the popup dialog allows you to enter this “Starting Address,” as shown in [Figure 4-4](#)

Figure 4-4 New ModbusAsyncDevice popup dialog



Note this dialog allows you to add a sequentially-addressed range of *multiple* ModbusAsyncDevices, by setting “Number to Add” more than 1, with the first device using the starting (Modbus) address. This technique may be useful in cases where all devices in that range will have a different set of proxy points.

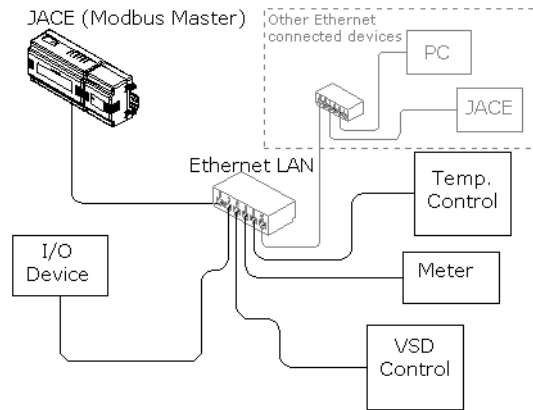
However, in cases where you have “like devices,” you might create a single ModbusAsyncDevice first, configure its proxy points and other components as needed, and then *duplicate* it as many times as needed. Then you can change the “Device Address” of each duplicate to a unique number, as needed.

When you click **OK** in the dialog shown in [Figure 4-4](#), the next New dialog provides a number of “standard” new device properties to enter (such as Name and Enabled), and others common to all client Modbus devices (ModbusAsyncDevice, ModbusTcpDevice, ModbusTcpGatewayDevice). For more details on these Modbus-related properties, see “[About Modbus client devices](#)” on page 4-12.

About Modbus TCP networks

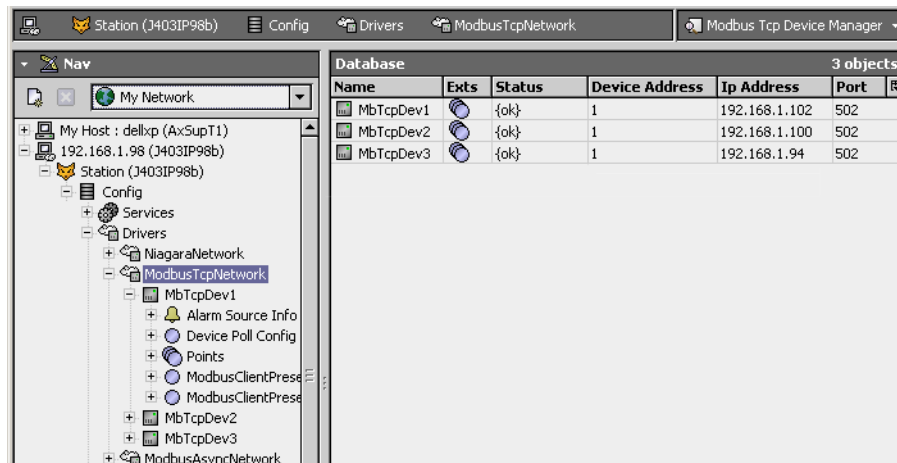
A ModbusTCPNetwork automatically binds to the TCP/IP setup of the host JACE platform's Ethernet adapter. Again, the JACE appears as the Modbus master on a network of Modbus TCP slave devices, however network connectivity is Ethernet/IP (see Figure 4-2).

Figure 4-5 ModbusTcpNetwork with Modbus TCP slave devices



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

Figure 4-6 ModbusTcpNetwork in JACE station



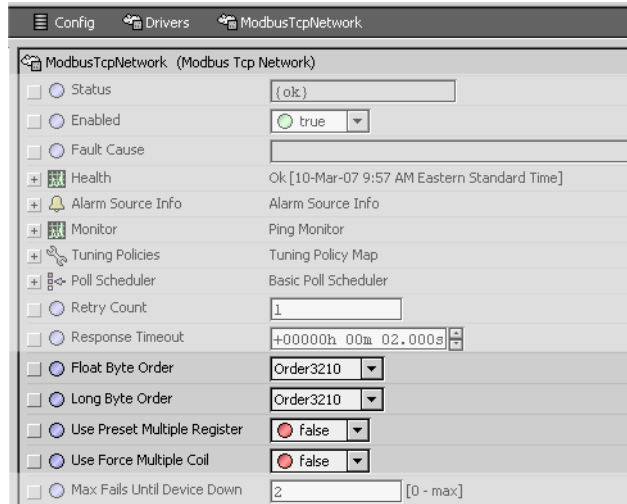
As shown in Figure 4-6, each child device is represented by a ModbusTcpDevice, and has a unique IP address, as well as other Modbus config data and starting addresses for Modbus data items (coils, inputs, input registers, holding registers). There are typically many child ModbusTcpDevices.

Modbus Tcp Network configuration is straightforward, using the property sheet of the network object.

Modbus Tcp Network configuration

In the property sheet of the ModbusTcpNetwork (Figure 4-7), you review the default global values for Modbus device data (the network automatically binds to the local TCP/IP address of the JACE).

Figure 4-7 ModbusTcpNetwork property sheet



Note the ModbusTcpNetwork has the standard collection of network components, such as for status, health, monitor, tuning policies, and poll scheduler. For more details, see “Common network components” in the *Drivers Guide*.

In addition, the following properties have special importance:

- “Global” Modbus device defaults, which (as needed) can be overridden in the [Modbus Config \(client device level\)](#) slot of any child ModbusTcpDevice:
 - **Float Byte Order**
Specifies the byte-order used to assemble or receive floating-point (32-bit) values in messages.
 - **Long Byte Order**
Specifies the byte-order used to assemble or receive long integer (32-bit) values in messages.
 - **Use Force Multiple Coil**
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.
 - **Use Preset Multiple Register**
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.

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

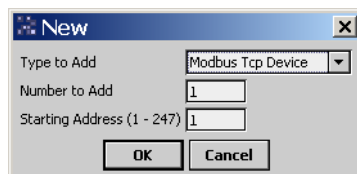
Double-click the ModbusTcpNetwork for the default **Modbus Tcp Device Manager** view, which you can use to add new ModbusTcpDevice children. For general information, see “About the Device Manager” in the *Drivers Guide*. See “[Modbus Tcp Device Manager notes](#)” for additional details.

Modbus Tcp Device Manager notes

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”.

However, when using the Modbus Tcp Device Manager view to create new devices, the initial popup dialog allows you to enter this Modbus “Starting Address”, as shown in [Figure 4-8](#), and you enter the IP address in the subsequent dialog.

Figure 4-8 New ModbusTcpDevice popup dialog



Note the initial dialog allows you to add a sequentially-addressed range of *multiple* ModbusTcpDevices, by setting “Number to Add” more than 1, with the first device using the starting (Modbus) address. This technique is probably more useful in a ModbusAsyncNetwork, where devices do not use IP addressing. It is recommended you leave the “Device Address” at 1 for all ModbusTcpDevice objects, unless vendor’s documentation for Modbus TCP devices state otherwise.

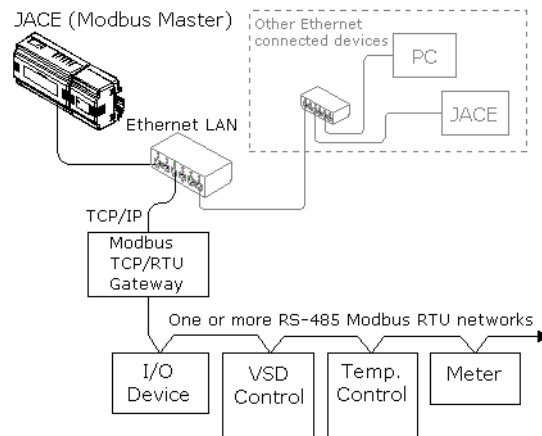
Note in cases where you have “like devices,” you might create a single ModbusTcpDevice first, configure its proxy points and other components as needed, and then *duplicate* it as many times as needed. Then you can change the “Ip Address” property of each duplicate to the IP address in use, as needed.

When you click **OK** in the initial dialog shown in Figure 4-8, the next New dialog provides a number of “standard” new device properties to enter (such as Name and Enabled), and others common to all client Modbus devices (ModbusAsyncDevice, ModbusTcpDevice, ModbusTcpGatewayDevice). For more details on these Modbus-related properties, see “About Modbus client devices” on page 4-12.

About Modbus TCP Gateway networks

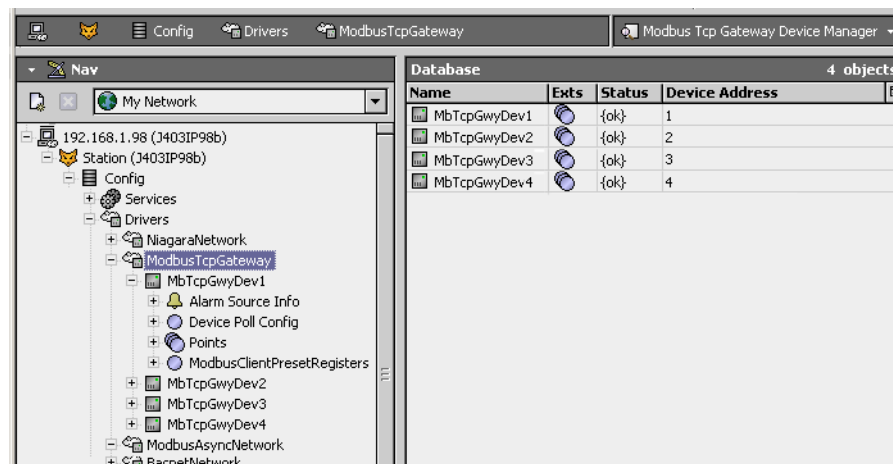
A ModbusTCPGateway is a network-level object that also represents a particular *device*: a Modbus TCP-to-Serial gateway, where this device has an IP address reachable by the station. On the gateway’s “far side” are serially-connected Modbus devices (typically Modbus RTU via RS-485), as shown in Figure 4-9. Those serial Modbus devices (RTU or ASCII) are represented by child devices under the gateway (network) object.

Figure 4-9 ModbusTcpGateway models Modbus TCP/Serial gateway and serial Modbus devices



In addition to the IP address and TCP port used by the gateway, there are global properties on the network’s property sheet specific to Modbus, for example the default order for “float” and “long” numeric data (overrideable within each child device).

Figure 4-10 ModbusTcpGateway in JACE station



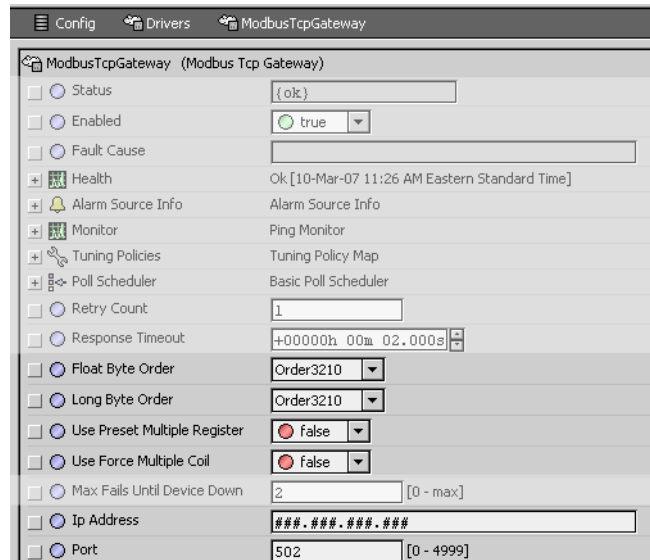
The station acts as Modbus master to the serially-connected Modbus devices on the gateway’s far side. Each child device is represented by a ModbusTcpGatewayDevice, and has a unique Modbus address (1 to 247), as well as other Modbus config data and starting addresses for Modbus data items (coils, inputs,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

[Modbus Tcp Gateway configuration](#) is straightforward, using the property sheet of the network object.

Modbus Tcp Gateway configuration

In the property sheet of the ModbusTcpGateway (Figure 4-11), you configure the TCP/IP address used to connect to the Modbus gateway, and review other default network values.

Figure 4-11 ModbusTcpGateway property sheet



Note the ModbusTcpGateway has the standard collection of network components, such as for status, health, monitor, tuning policies, and poll scheduler. For more details, see “Common network components” in the *Drivers Guide*.

In addition, the following properties have special importance:

- “Global” Modbus device defaults, which (as needed) can be overridden in the [Modbus Config \(client device level\)](#) slot of any child ModbusTcpGatewayDevice:
 - **Float Byte Order**
Specifies the byte-order used to assemble or receive floating-point (32-bit) values in messages.
 - **Long Byte Order**
Specifies the byte-order used to assemble or receive long integer (32-bit) values in messages.
 - **Use Force Multiple Coil**
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.
 - **Use Preset Multiple Register**
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.
- **Ip Address**
Specifies the IP address of the “TCP/Ethernet-side” of the Modbus TCP/serial gateway. Must be unique from all other devices on the IP network. The default IP address is “###.###.###.###”, meaning that no IP address is assigned. Initially this as a blank placeholder—enter the IP address used to reach the TCP side of the Modbus gateway.
- **Port**
Specifies the TCP port used by Modbus message transactions. 502 is the “standard” Modbus TCP port. Leave at the default (502) unless the “TCP/Ethernet-side” of the Modbus TCP/serial gateway uses another TCP port.

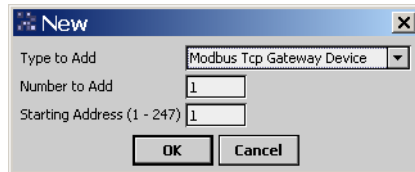
Other ModbusTcpGateway properties such as “Retry Count”, “Response Timeout”, and “Max Fails Until Device Down” are typically left at defaults, unless particular reasons dictate change.

Double-click the ModbusTcpGateway for the default **Modbus Tcp Gateway Device Manager** view, which you can use to add new ModbusTcpGatewayDevice children. For general information, see “About the Device Manager” in the *Drivers Guide*. See “[Modbus Tcp Gateway Device Manager notes](#)” for additional details.

Modbus Tcp Gateway Device Manager notes

Each ModbusTcpGatewayDevice you add under a ModbusTcpGateway requires the unique Modbus address (1–247) used by that device on serial “far-side” of that gateway. When using the Modbus Tcp Gateway Device Manager view to create new devices, the popup dialog allows you to enter this “Starting Address”, as shown in [Figure 4-12](#).

Figure 4-12 New ModbusTcpGatewayDevice popup dialog



Note this dialog allows you to add a sequentially-addressed range of *multiple* ModbusTcpGatewayDevices, by setting “Number to Add” more than 1, with the first device using the starting (Modbus) address. This may be useful in cases where all devices in that range will have a different set of proxy points.

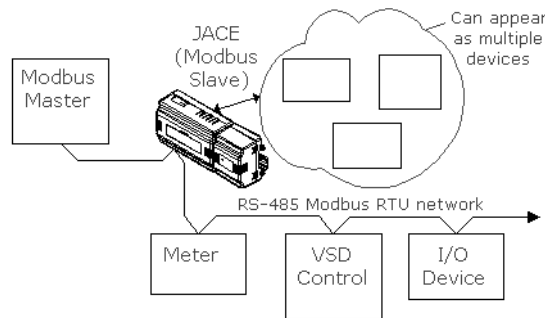
However, in cases where you have “like devices,” you might create a single ModbusTcpGatewayDevice first, configure its proxy points and other components as needed, and then *duplicate* it as many times as needed. Then you can change the “Device Address” of each duplicate to a unique number, as needed.

When you click **OK** in the dialog shown in [Figure 4-12](#), the next New dialog provides a number of “standard” new device properties to enter (such as Name and Enabled), and others common to all client Modbus devices (ModbusAsyncDevice, ModbusTcpDevice, ModbusTcpGatewayDevice). For more details on these Modbus-related properties, see “[About Modbus client devices](#)” on page 4-12.

About Modbus Slave networks

Each ModbusSlaveNetwork requires a serial port on the host JACE platform. The JACE connects on the Modbus RTU or ASCII network and functions as another Modbus slave (server). If needed, you can make it appear as *multiple* Modbus devices, by adding multiple ModbusSlaveDevices under the ModbusSlaveNetwork, and assigning each one a unique Modbus address (1–247) for that network.

Figure 4-13 ModbusSlaveNetwork with JACE as slave, here on an RS-485 Modbus RTU network



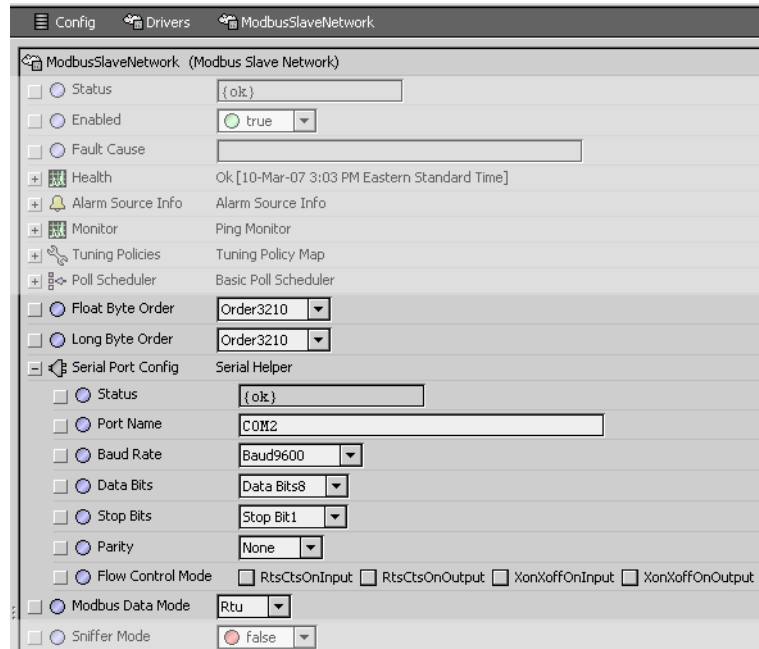
The station acts as a Modbus slave (server) to queries received from a serially-connected Modbus master device. In each uniquely-addressed ModbusSlaveDevice, you specify the ranges for available Modbus data items (coils, inputs, input registers, holding registers). In some cases, only a single child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

[Modbus Slave Network configuration](#) is straightforward, using the property sheet of the network object.

Modbus Slave Network configuration

In the property sheet of the ModbusSlaveNetwork (Figure 4-14), you configure the serial port/comm settings needed to communicate to the attached Modbus master, and review other default values.

Figure 4-14 ModbusSlaveNetwork property sheet



Note the ModbusSlaveNetwork has the standard collection of network components, such as for status, health, monitor, tuning policies, and poll scheduler. For more details, see “Common network components” in the *Drivers Guide*.

In addition, the following properties have special importance:

- “Global” Modbus device defaults, which (as needed) can be overridden in the [Modbus Config \(client device level\)](#) slot of any child ModbusSlaveDevice:
 - **Float Byte Order**
Specifies the byte-order used to assemble or receive floating-point (32-bit) values in messages.
 - **Long Byte Order**
Specifies the byte-order used to assemble or receive long integer (32-bit) values in messages.
- **Note:** *By default Niagara supports Modbus function codes 15 and 16 (Force Multiple Coils, Preset Multiple Registers), so you do not see these selections like you do in a client Modbus network.*
- **Serial Port Config**
The container slot in which you specify the serial port/communications setup required to talk to attached serial Modbus master.
- **Modbus Data Mode**
As either Rtu (the default) or Ascii, depending on the type of networked Modbus master.

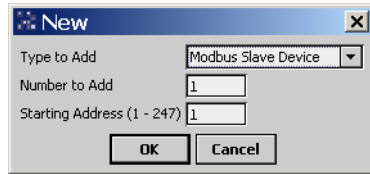
The other ModbusSlaveNetwork property “Sniffer Mode” is typically left at default, unless particular reasons dictate change.

Double-click the ModbusSlaveNetwork for the default **Modbus Slave Device Manager** view, which you can use to add one or more ModbusSlaveDevice children. For general information, see “About the Device Manager” in the *Drivers Guide*. See “[Modbus Slave Device Manager notes](#)” for additional details.

Modbus Slave Device Manager notes

Each ModbusSlaveDevice you add under a ModbusSlaveNetwork requires a Modbus address (1–247) unique from any other physical device on that network. When using the Modbus Slave Device Manager view to create new devices, the popup dialog allows you to enter this “Starting Address”, as shown in [Figure 4-15](#).

Figure 4-15 New ModbusSlaveDevice popup dialog



Note this dialog allows you to add a sequentially-addressed range of *multiple* ModbusSlaveDevices, by setting “Number to Add” more than 1, with the first device using the starting (Modbus) address. This technique may be useful in cases where you wish the station to appear as “multiple physical Modbus devices” (each with a different set of proxy points).

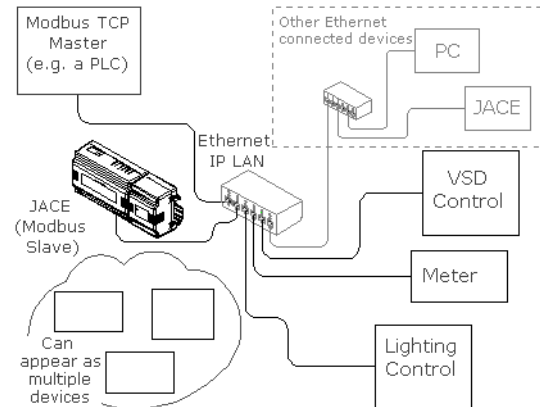
Or, you may wish to have the station appear simply as a single device, in which case you add only one ModbusSlaveDevice. Note that you can specify many ranges of Modbus data items (coils, inputs, input registers, holding registers) in any or all ModbusSlaveDevice components.

When you click **OK** in the dialog shown in [Figure 4-15](#), the next New dialog provides a few “standard” new device properties to enter (such as Name and Enabled), and others common to all server Modbus devices (ModbusSlaveDevice, ModbusTcpSlaveDevice). For more details on these Modbus-related properties, see [“About Modbus client devices”](#) on page 4-12.

About Modbus TCP Slave networks

A ModbusTCPSlaveNetwork automatically binds to the TCP/IP setup of the host JACE platform’s Ethernet adapter, assuming the IP address of the station. The JACE functions as a Modbus slave (server). If needed, it can appear as *multiple* Modbus slaves, by adding multiple ModbusTcpSlaveDevices under the ModbusSlaveNetwork, and assigning each one a unique Modbus address (1–247). However, all slaves are reachable only via the IP address of the host JACE platform.

Figure 4-16 ModbusTcpSlaveNetwork with JACE as slave

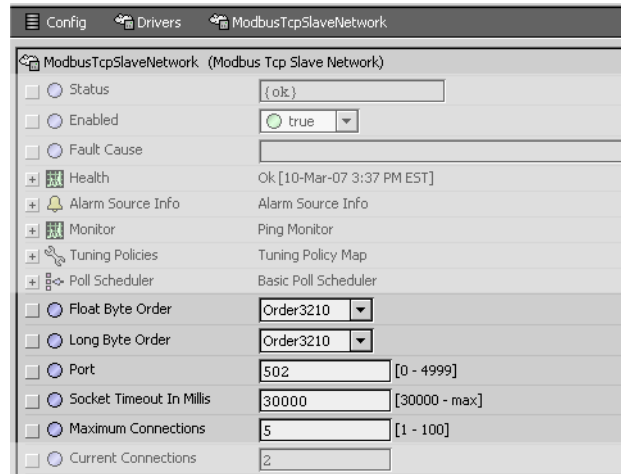


The station acts as a Modbus slave (server) to queries received from a Modbus TCP master device. Each uniquely-addressed child device is represented by a ModbusTcpSlaveDevice, in which you specify ranges for available Modbus data items (coils, inputs, input registers, holding registers). In some cases, only a single child ModbusTcpSlaveDevice may exist to represent the station.

Modbus Tcp Slave Network configuration

In the property sheet of the ModbusTcpSlaveNetwork (Figure 4-17), you review the default global values for Modbus device data, and specify other TCP connection settings.

Figure 4-17 ModbusTcpSlaveNetwork property sheet



Note the ModbusTcpSlaveNetwork has the standard collection of network components, such as for status, health, monitor, tuning policies, and poll scheduler. For more details, see “Common network components” in the *Drivers Guide*.

In addition, the following properties have special importance:

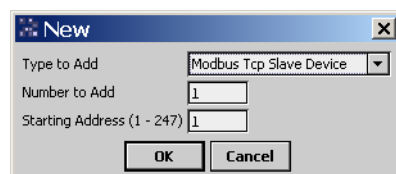
- “Global” Modbus device defaults, which (as needed) can be overridden in the [Modbus Config \(server device level\)](#) slot of any child ModbusSlaveDevice:
 - **Float Byte Order**
Specifies the byte-order used to assemble or receive floating-point (32-bit) values in messages.
 - **Long Byte Order**
Specifies the byte-order used to assemble or receive long integer (32-bit) values in messages.
- **Note:** *By default Niagara supports Modbus function codes 15 and 16 (Force Multiple Coils, Preset Multiple Registers), so you do not see these selections like you do in a client Modbus network.*
- **Port**
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**
Default is 30000 (milliseconds, or 30 seconds—the minimum). You can adjust upwards if necessary.
- **Maximum Connections**
Default is 5. You can adjust from 1 to 100 as needed.

Double-click the ModbusTcpSlaveNetwork for the default **Modbus Tcp Slave Device Manager** view, which you can use to add one or more ModbusTcpSlaveDevice children. For general information, see “About the Device Manager” in the *Drivers Guide*. See “[Modbus Tcp Slave Device Manager notes](#)” for additional details.

Modbus Tcp Slave Device Manager notes

Each ModbusTcpSlaveDevice you add under a ModbusTcpSlaveNetwork may have a Modbus address (1–247), although typically a Modbus TCP device uses only its unique **IP address**. When using the Modbus Tcp Slave Device Manager view to create new devices, the popup dialog allows you to enter this Modbus “Starting Address”, as shown in Figure 4-18.

Figure 4-18 New ModbusTcpSlaveDevice popup dialog



Note this dialog allows you to add a sequentially-addressed range of **multiple** ModbusTcpSlaveDevices, by setting “Number to Add” more than 1, with the first device using the starting (Modbus) address. This technique may be useful in cases where you wish the station to appear as “multiple physical Modbus TCP devices” (each with a different set of proxy points)—however, note all devices are reached only through the station’s (same) host IP address.

More typically, you may wish to have the station appear simply as a single device, in which case you add only **one** ModbusTcpSlaveDevice. Note that you can specify many ranges of Modbus data items (coils, inputs, input registers, holding registers) in any or all ModbusTcpSlaveDevice components.

When you click **OK** in the dialog shown in [Figure 4-18](#), the next New dialog provides a few “standard” new device properties to enter (such as Name and Enabled), and others common to all server Modbus devices (ModbusSlaveDevice, ModbusTcpSlaveDevice). For more details on these Modbus-related properties, see “[About Modbus server devices](#)” on page 4-26.

About Master (client) types

A master-type Modbus network allows the station to regularly poll slave Modbus devices with **client** requests, providing Modbus data. Proxy points under each device in the network are Modbus “client” types. 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.

See the following sections:

- [About Modbus client devices](#)
- [About Modbus client proxy points](#)
- [About Modbus preset components](#)
- [About Modbus \(client\) file records](#)

About Modbus client devices

Modbus client devices include the ModbusAsyncDevice, ModbusTcpDevice, and ModbusTcpGatewayDevice. Each of these device components actually represents a remote Modbus **slave**, that is, a remote device that listens for Modbus queries from a Modbus master (JACE’s station), and sends responses.

All three types of Modbus client devices are very similar, having a single frozen **Points** device extension, with the default Modbus Client Point Manager view. The same type of [Modbus client proxy points](#) are used under any Points device extension, as well as any of the “preset” and “file record” objects.

Note: *Each device type is specific to a particular parent network type—for example, you cannot copy a ModbusAsyncDevice under a ModbusTcpGateway, or a ModbusTcpGatewayDevice under a ModbusTcpNetwork. This is not a problem when working in the device manager for any of the three client networks, as the “New” function (to add devices) automatically selects the proper child device component.*

In addition to common device slots (see “Common device components” in the *Drivers Guide*), all three types of Modbus client devices have similar properties for Modbus configuration. 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.

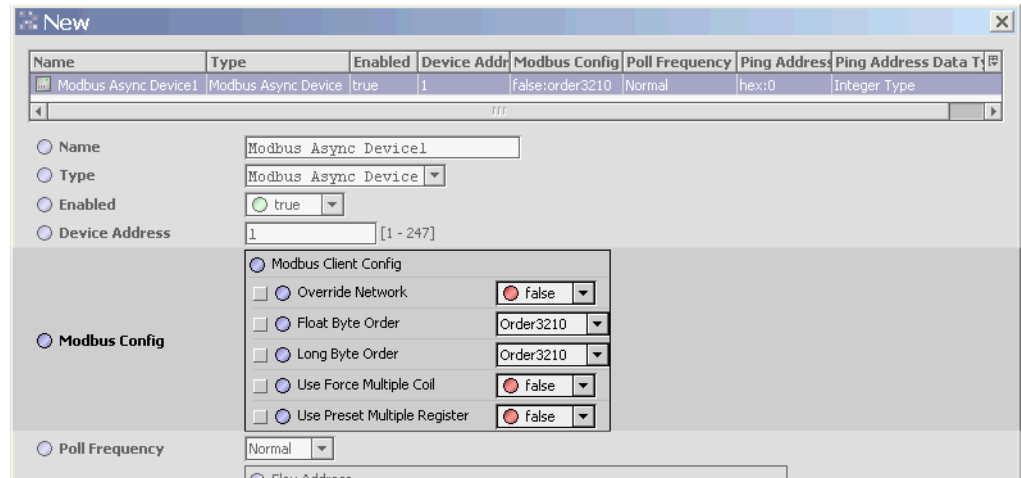
The following sections provide more details on these Modbus device configuration properties:

- [Modbus Config \(client device level\)](#)
- [Ping Address properties](#)
- [Base Address properties](#)
- [Device Poll Config](#)

Modbus Config (client device level)

Each Modbus client device has a “Modbus Config” container slot with 5 properties—you can access them on the device’s property sheet, as well as the New or Edit dialog for a device object when in the device manager view (of its parent network). [Figure 4-19](#) shows the properties in the New dialog for a device.

Figure 4-19 Modbus Config properties in New/Edit dialog when working in network's device manager view



These properties allow you to “override” the network-level (global) Modbus Config equivalent settings for handling Modbus data from and to this device, and are described as follows:

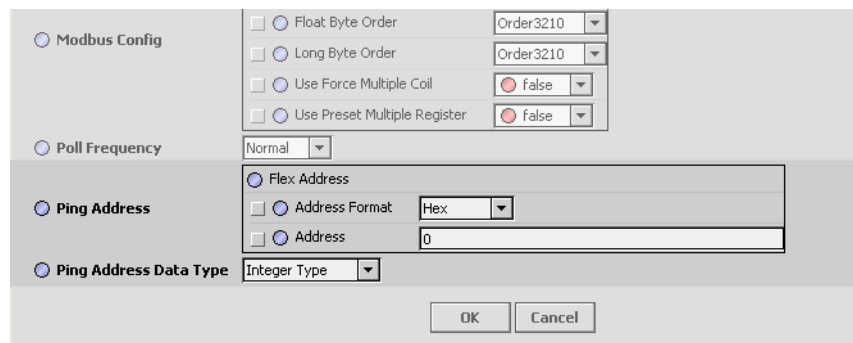
- **Override Network**
 The default, false, means the network-level settings are used. Set this to true whenever you want the settings below to be used instead of the equivalent network-level values.
- **Float Byte Order**
 Specifies the byte-order used to assemble or receive floating-point (32-bit) values in messages.
- **Long Byte Order**
 Specifies the byte-order used to assemble or receive long integer (32-bit) values in messages.
- **Use Force Multiple Coil**
 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.
- **Use Preset Multiple Register**
 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.

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

Ping Address properties

Each Modbus client device has a “Ping Address” container slot with 3 properties—you can access them on the device’s property sheet, as well as the New or Edit dialog for a device object when in the device manager view (of its parent network). [Figure 4-20](#) shows ping properties in the New dialog for a device.

Figure 4-20 Ping Address properties in New/Edit dialog when working in network's device manager view



These properties specify a particular data address (either input register or holding register) to use as the device status test (meaning “Monitor” ping requests). Ping requests are generated at the network-level by the configurable network monitor. See “About Monitor” in the *Drivers Guide* for more details.

When enabled, a network’s monitor periodically pings (queries) this address. While receiving any response from the device, including an exception response, this is considered proof of communication, and the Modbus client device is no longer considered “down” if it had been previously marked “down”.

The device ping address properties are:

- **Ping Address**
 - Address Format — either Hex (default), Decimal, or Modbus
 - Address — numerical address, expressed in the selected format (0 is default).
See “Modbus data addresses” on page 3-2 for general information.
- **Ping Address Data Type**
To specify one of the four numeric data types, either Integer (the default), Long, Float, or Signed Integer type. See “Numerical data types” on page 3-6 for general information
- **Ping Address Reg Type**
To specify either a Holding register (the default) or an Input register

Note: *Default device ping address values (first holding register) typically “work,” as even an exception response from the device is considered OK for device status. While this is considered proof of communications, it is recommended to configure the ping address to a **known** data address and type according to the device’s documentation, to ensure device status monitoring **without** an exception response.*

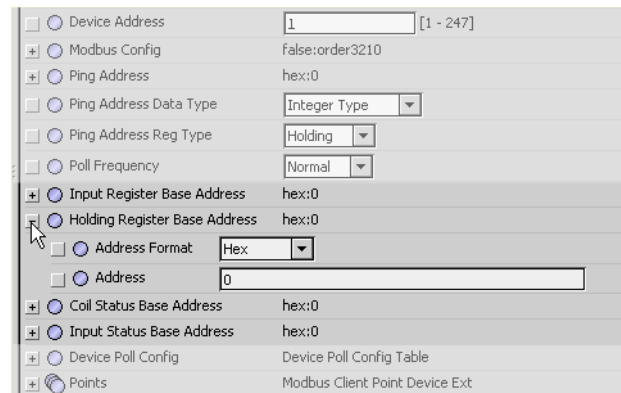
Base Address properties

Each Modbus client device has four container slots for setting a “base address” on each of the four data item types (coils, inputs, holding registers, input registers), as follows:

- Input Register Base Address
- Holding Register Base Address
- Coil Status Base Address
- Input Status Base Address

Access these slots on the property sheet of the client Modbus device, as shown in [Figure 4-21](#).

Figure 4-21 Base Address slots of client Modbus device



These properties specify an “address offset” that is **added** to any child proxy point using a data address for that data item type. Base addressing of coils and holding registers also affects addressing of “Preset” components (see “About Modbus preset components” on page 4-21).

Each base address container slot has two properties:

- Address Format — either Hex (default), Decimal, or Modbus (do **not** select Modbus, see [Note](#)).
- Address — numerical address, expressed in the selected format (0 is default).

For example, if a device’s Holding Register Base Address is set to “Decimal, 100”, any child Modbus proxy point using a holding register “Data Address” of “Decimal, 13” is effectively addressed as “Decimal, 113” (Absolute Address). See “Modbus client point ProxyExt properties” on page 4-19 for related details.

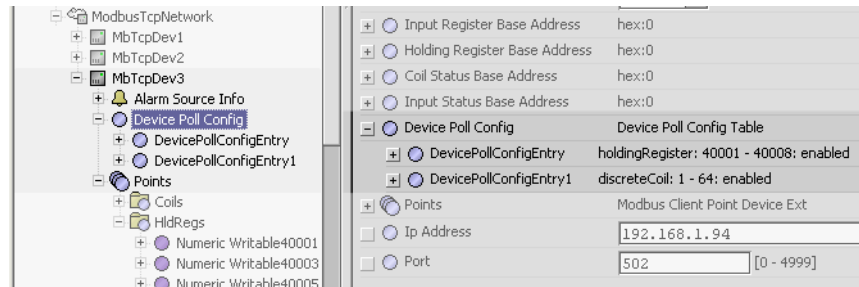
Typically, all Base Address properties of a device are left at default (hex: 0). However, you can use them as an engineering method (along with multiple device objects) if a Modbus device has data partitioned into multiple areas with repeating address patterns. This way, the same device (and child proxy points) can be replicated, and the only address changes made in the “Base Address” offsets.

Note: *If entering Base Addresses in a device, Modbus formatted data addresses cannot be used (select Hex or Decimal)—and this also applies to Data Address properties of child proxy points.*

Device Poll Config

Each Modbus client device has a frozen container slot “Device Poll Config,” which you can see under the device when expanded in the Nav tree, as well as in the device’s property sheet, as shown in Figure 4-22.

Figure 4-22 Device Poll Config container of client Modbus device



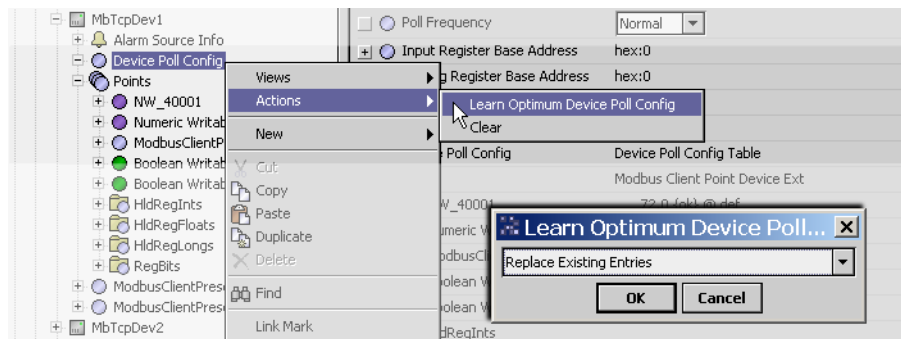
By default (initially) the Device Poll Config container is *empty*, but it can hold one or more “Device Poll Config Entry” children, which configure and enable device polling. A device poll permits a single NiagaraAX Modbus query message to retrieve a number of consecutive data values. Device-level polling may help overall polling efficiency by reducing the number of polls necessary at the point-level.

Note: *In a few cases, a device-level poll has actually proven counterproductive, at least for improving polling efficiency. It was determined that the target Modbus device was taking 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.*

Configuring Device Poll Config Based upon the existing child Modbus proxy points, you can configure “device-level” polls (Device Poll Config) using either of these two methods:

- Automatically, using the right-click “Learn Optimum Device Poll Config” **action** on the Device Poll Config container (Figure 4-23). This creates one or more Device Poll Config Entry components, already configured to produce the needed device polls to query consecutively addressed data items.
Note: *The station must be running to use this method (offline station usage not supported).*
- Manually, by editing properties in any existing Device Poll Config Entry, and duplicating/re-editing it (or by copying entries from the modbusAsync or modbusTcp palette and editing as necessary).

Figure 4-23 Action “Learn Optimum Device Poll Config” dynamically adds/configures child components



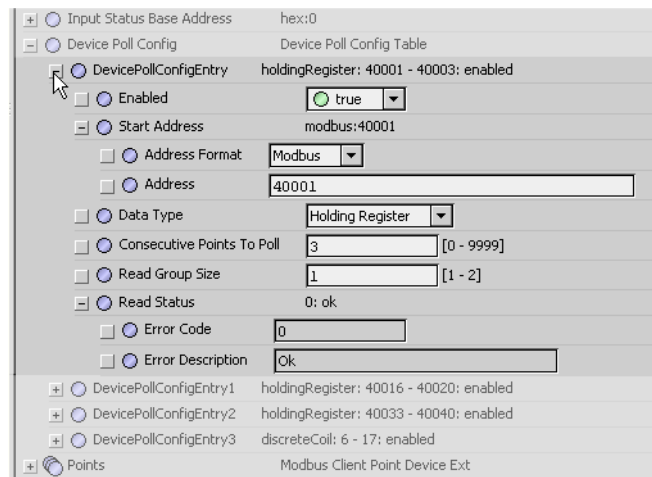
Typically, you choose the automatic (action) method—note it lets you **replace** existing device poll entries (start over), or **append** to the existing device polls. The Device Poll Config container also has a separate “Clear” action you can use to remove all existing Device Poll Config Entry children.

Note: *For best results, it is recommended that you first create all needed proxy points under a device, before configuring for device polling.*

When executing the learn action, its algorithm looks for any consecutively addressed Modbus proxy points, and creates a DevicePollConfigEntry if 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. Remember you can always create, configure, and remove DevicePollConfigEntries until you find the most efficient device-polling scheme.

Device Poll Config Entry Figure 4-24 shows a Device Poll Config Entry child expanded, and its properties that were populated from the learn action on the device's **Device Poll Config** slot.

Figure 4-24 Expanded Device Poll Config Entry, representing one device poll message



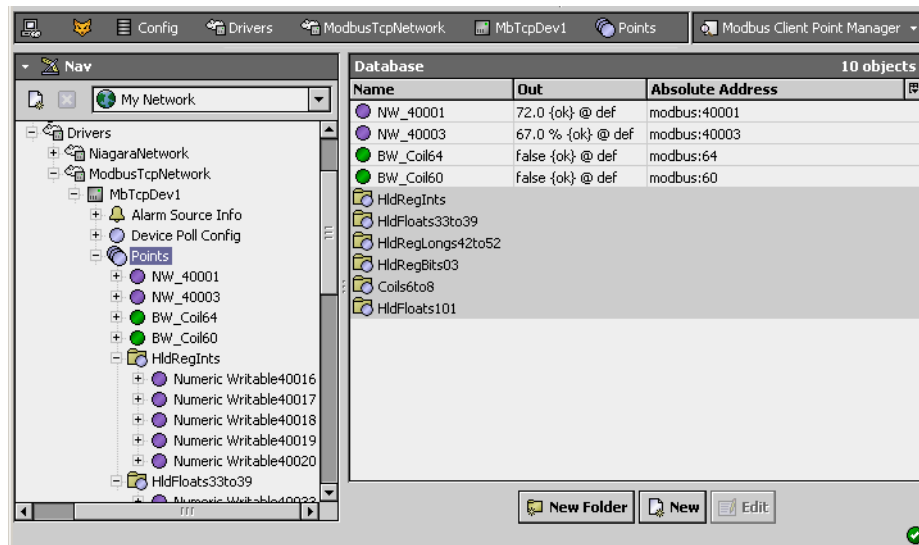
Properties of a Device Poll Config Entry child are described as follows:

- **Enabled** — By default, is true. If set to false, associated proxy points use individual point polls instead.
- **Start Address** — First data item address, including format and numerical address.
- **Data Type** — Modbus data type: Holding Register, Input Register, Discrete Coil, or Discrete Input
- **Consecutive Points to Poll** — Number of data items to poll starting from start address
- **Read Group Size** — 1 or 2, Usually 1 unless all data items are 2-register types (float or long values), although a 1 works the same, providing “consecutive points to poll” is really consecutive registers.
- **Read Status** — A numerical error code 0–2, and a corresponding text description.

Modbus Client Point Manager notes

The Modbus Client Point Manager is the default view of the **Points** extension under any client Modbus device (ModbusAsyncDevice, ModbusTcpDevice, ModbusTcpGatewayDevice), as shown in Figure 4-27.

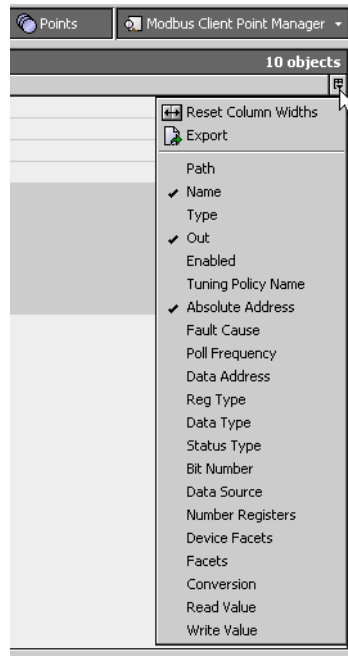
Figure 4-25 Modbus Client Point Manager view is default view for Points under client Modbus device



This view is also the default for any “points folder” created with the **New Folder** button, and operates like most point manager views for NiagaraAX drivers—see the section “About the Point Manager” in the **Drivers Guide** for general details. Note that due to the simplicity of the Modbus protocol, there is no “Discover, Add, and Match” (Learn process). Instead you make proxy points using the **New** button, after studying the vendor’s documentation for Modbus data in each Modbus device. See “About Modbus client proxy points” on page 4-17 for more details.

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, as shown in [Figure 4-26](#).

Figure 4-26 Table Options menu allows you to select additional and/or different data columns

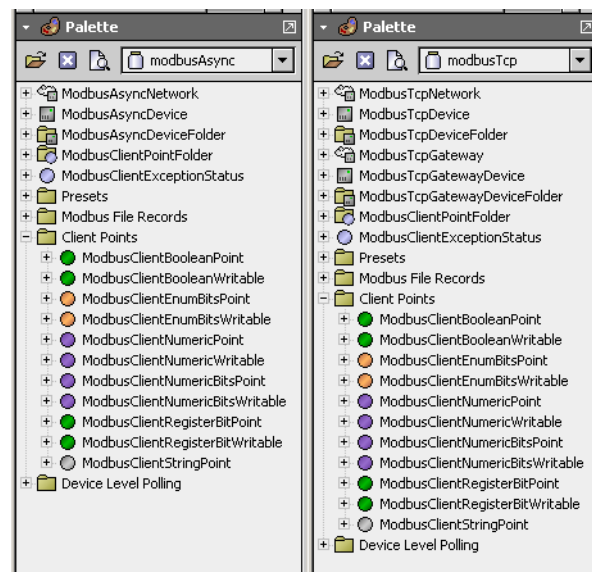


For example, during the configuration process you may wish to see “Fault Cause” and “Data Source.”

About Modbus client proxy points

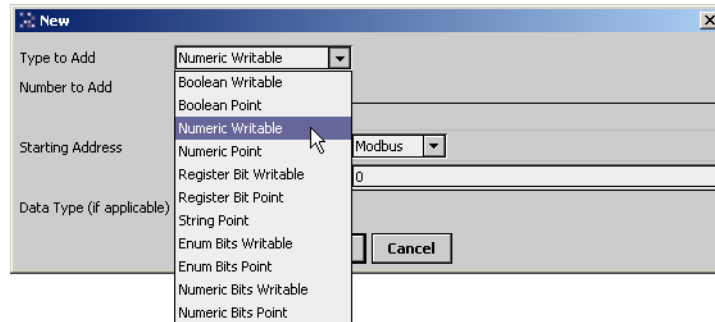
Modbus client proxy points are similar to other driver’s proxy points. See “About proxy points” in the *Drivers Guide* for general details. Note that the *same collection* of client proxy points is used in devices under a ModbusAsync, ModbusTcp, and ModbusTcpGateway network—you can find them in the “Client Points” folder in either palette type (modbusAsync or modbusTcp), as shown in [Figure 4-27](#).

Figure 4-27 Client Points folder is same in modbusAsync palette and modbusTcp palette



Although sometimes you may need to copy components from the palette, note that the same selection of client point types is available in the **New** dialog, when adding points in the [Modbus Client Point Manager](#) view of a device (of its Points extension), as shown in [Figure 4-28](#).

Figure 4-28 New dialog from Add in Modbus Client Point Manager provides client point selection



Note: The last four types listed, Enum Bits Writable through Numeric Bits Point, require Modbus 3.5.26 or higher, or AX-3.6 or later.

Typically, this is the quickest way to add Modbus client proxy points, because you can specify a number of points if consecutively addressed. See “Consecutive address usage (NiagaraAX)” on page 3-4.

The following sections provide additional details on client Modbus proxy points:

- [Types of Modbus client proxy points](#)
- [Modbus client point ProxyExt properties](#)

Types of Modbus client proxy points

You can select from the following Modbus client proxy point types:

- [Boolean Writable](#)
To read/write a Modbus coil.
- [Boolean Point](#)
To read either a Modbus coil or an input.
- [Numeric Writable](#)
To read/write a Modbus holding register value.
Note you must specify the Data Type, as either integer, long, float, or signed integer.
- [Numeric Point](#)
To read either a Modbus holding register value or an input register value.
Note you must specify the Data Type, as either integer, long, float, or signed integer.
- [Register Bit Writable](#)
To read/write a specific bit in a Modbus holding register (select Bit Number in setup).
- [Register Bit Point](#)
To read a specific bit in either a Modbus holding register or an input register (select Bit Number in setup).
- [String Point](#)
To read 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.

Note: Remaining client point types listed below are available only in Modbus 3.5.26 or higher, or AX-3.6 or later. See the subsequent [Example](#) for a possible application.

- [Enum Bits Writable](#)
To read or write some number of consecutive bits within a holding register, with the resulting integer as the out (ordinal) value of the Enum Writable.
- [Enum Bits Point](#)
To read some number of consecutive bits within a holding or input registering the resulting integer as the out (ordinal) value of the Enum Point.
- [Numeric Bits Writable](#)
To read or write some number of consecutive bits within a holding registering the resulting integer as the out value of the Numeric Writable.
- [Numeric Bits Point](#)
To read some number of consecutive bits within a holding or input registering the resulting integer as the out value of the Numeric Point.

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.

Modbus client point ProxyExt properties

Apart from the standard “core” proxy extension properties (see “ProxyExt properties” in the *Drivers Guide*), these ProxyExt properties have special importance in Modbus client proxy points:

- **Fault Cause**
(Read only) If the point is in fault from an exception response received from the slave device, that exception string appears here. For example: “Read fault: illegal data address”. See “[Exception codes](#)” on page 3-10 for more details.
- **Read Value**
(Read only) Shows last polled value as well as the state, for example “71 {ok}” or “false {ok}”.
- **Write Value**
(Read Only) Shows the last written value, as well as the state and priority level, for example “70 {overridden} @8”.
- **Data Address**
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:
 - Address Format — either Modbus (default), Hex, or Decimal
 - Address — numerical address, expressed in the selected format.See “[Modbus data addresses](#)” on page 3-2 for general information.
For example, the following are all equivalent addresses:
 - Modbus, 40012
 - Hex, 0B
 - Decimal, 11*Note: If using Hex or Decimal format, for most read-only points you need to specify the “Reg Type” property, to clarify whether holding register or input register. For related details, see “[Data address format in NiagaraAX](#)” on page 3-5.*
- **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 — as selected, either Modbus (default), Hex, or Decimal
 - Address — numerical address, expressed in the selected Data Address format.
- **Data Source**
(Read only) Displays the type of polling used to collect the value. If it shows “Point Poll” then the Absolute Address is not contained within the range of addresses specified in a Device Poll Config Entry—thus a separate, individual, point-level poll message is required to collect the value for this Modbus proxy point (at a poll rate specified by the ‘Poll Frequency’ property).
If it displays “Device Poll” then the Absolute Address is contained within the range of addresses specified in a Device Poll Config Entry, and thus the value for this Modbus proxy point is “picked” from the device-level polling.

Depending on the *type* of Modbus proxy point, one or more *additional* properties are used in its ProxyExt to clarify or confirm the data item needed, as follows:

- [ModbusClientBooleanPoint](#)
- [ModbusClientBooleanWritable](#)
- [ModbusClientNumericPoint](#)
- [ModbusClientNumericWritable](#)
- [ModbusClientRegisterBitPoint](#)
- [ModbusClientRegisterBitWritable](#)
- [ModbusClientEnumBitsPoint](#)
- [ModbusClientEnumBitsWritable](#)
- [ModbusClientNumericBitsPoint](#)
- [ModbusClientEnumBitsWritable](#)

ModbusClientBooleanPoint Or “Boolean Point”, has the following in addition to other [Modbus client point ProxyExt properties](#):

- **Status Type**
Coil or Input. Specifies whether the point is an Input status type or a Coil status type. Selections only apply if the “Data Address” format used is Hex or Decimal. (The Modbus address format, if used, automatically sets this property value).

ModbusClientBooleanWritable Or “Boolean Writable”, has the following in addition to other [Modbus client point ProxyExt properties](#):

- **Status Type**
(Read only) Always Coil status type for a writable status item.

ModbusClientNumericPoint Or “Numeric Point”, has the following in addition to other [Modbus client point ProxyExt properties](#):

- **Reg Type**
Holding or Input. Specifies whether the value is read from an input register or holding register. Selections only apply if the “Data Address” format used is Hex or Decimal. (The Modbus address format, if used, automatically sets this property value).
- **Data Type**
Specifies the data type used by the associated data point. Integer and signed integer are 16-bit (single register) data types; long and float are 32-bit types (with the starting address specified in “Data Address”). Values for long and float selections are based upon the network’s byte-order config setup (or may be overridden at the device-level).

ModbusClientNumericWritable Or “Numeric Writable”, has the following in addition to other [Modbus client point ProxyExt properties](#):

- **Reg Type**
(Read only) Always Holding register for a writable numeric value.
- **Data Type**
Specifies the data type used by the associated data point. Integer and signed integer are 16-bit (single register) data types; long and float are 32-bit types (with the starting address specified in “Data Address”). Values for long and float selections are based upon the network’s byte-order config setup (or may be overridden at the device-level).

ModbusClientRegisterBitPoint Or “Register Bit Point”, has the following in addition to other [Modbus client point ProxyExt properties](#):

- **Reg Type**
Holding or Input. Specifies whether the bit is read from an input register or holding register. Selections only apply if the “Data Address” format used is Hex or Decimal. (The Modbus address format, if used, automatically sets this property value).
- **Bit Number**
Specifies the bit (numbered 0 - 15, least significant bit first) to read from the specified (16-bit) Modbus register. For example, if the specified register value was “0000000000001000”, setting the “Bit Number” to 3 would read a “1” (True).

ModbusClientRegisterBitWritable Or “Register Bit Writable”, has the following in addition to other [Modbus client point ProxyExt properties](#):

- **Reg Type**
(Read only) Always Holding register for a writable bit.
- **Bit Number**
Specifies the bit (numbered 0 - 15, least significant bit first) to write to the specified (16-bit) Modbus register. For example, if the specified register value was “0000000000000000”, setting the “Bit Number” to 3 and writing a True (“1”) would cause the specified register value to become “0000000000001000”.

ModbusClientStringPoint Or “StringPoint”, has the following in addition to other [Modbus client point ProxyExt properties](#):

- **Number Registers**
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 with a value of 4A41 hex (19009 decimal) returns String characters “JA,” where: 4A h = J and 41 h = A.

ModbusClientEnumBitsPoint Or “Enum Bits Point” (build 3.5.26 or higher, or AX-3.6 or later) has the following in addition to other [Modbus client point ProxyExt properties](#):

- **Reg Type**
Holding or Input. Specifies whether bits are read from an input register or holding register. Selections only apply if the “Data Address” format used is Hex or Decimal. (The Modbus address format, if used, automatically sets this property value).
- **Beginning Bit**
Specifies the first bit, 0 to 15, used in reading consecutive bits in the specified holding or input register, until “Number of Bits” have been read. The integer read is the out value (ordinal portion of the point’s facets).
- **Number of Bits**
Specifies the number of bits read in the register, starting with the “Beginning Bit”.

ModbusClientEnumBitsWritable Or “Enum Bits Writable” (build 3.5.26 or higher, or AX-3.6 or later) has the following in addition to other [Modbus client point ProxyExt properties](#):

- **Reg Type**
(Read only) Always Holding register for writable bits.
- **Beginning Bit**
Specifies the first bit, 0 to 15, used in reading or writing consecutive bits in the specified register, until “Number of Bits” have been read or written. The integer read (or written) is the out value (ordinal portion of the point’s facets).
- **Number of Bits**
Specifies the number of bits read or written in the register, starting with the “Beginning Bit”.

ModbusClientNumericBitsPoint Or “Numeric Bits Point” (build 3.5.26 or higher, or AX-3.6 or later) has the following in addition to other [Modbus client point ProxyExt properties](#):

- **Reg Type**
Holding or Input. Specifies whether bits are read from an input register or holding register. Selections only apply if the “Data Address” format used is Hex or Decimal. (The Modbus address format, if used, automatically sets this property value).
- **Beginning Bit**
Specifies the first bit, 0 to 15, used in reading consecutive bits in the specified holding or input register, until “Number of Bits” have been read. The combined integer value is the point’s out value.
- **Number of Bits**
Specifies the number of bits read in the specified register, starting with the “Beginning Bit”.

ModbusClientNumericBitsWritable Or “Numeric Bits Writable” (build 3.5.26 or higher, or AX-3.6 or later) has the following in addition to other [Modbus client point ProxyExt properties](#):

- **Reg Type**
(Read only) Always Holding register for writable bits.
- **Beginning Bit**
Specifies the first bit, 0 to 15, used in reading or writing consecutive bits in the specified register, until “Number of Bits” have been read or written. The combined integer value is the point’s out value.
- **Number of Bits**
Specifies the number of bits read or written in the register, starting with the “Beginning Bit”.

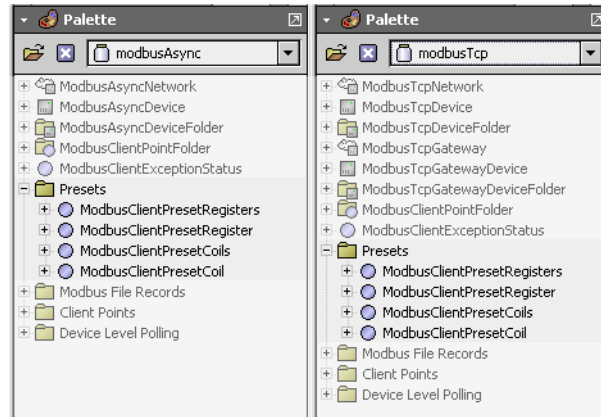
About Modbus preset components

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 into 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.

Preset components can be found in the modbusAsync palette and modbusTcp palette in the “Presets” folder, as shown in [Figure 4-29](#).

Note: *Modbus client preset components are not proxy points; there is no ProxyExt as data is not polled/read from the device—only written to it. Presets exist for both Modbus coils and holding registers—note these are the only two Modbus data items to which a Modbus master may write.*

Figure 4-29 Preset components in the modbusAsync and modbusTcp palettes



As needed, copy one or more of the following preset container components anywhere under a client Modbus device (ModbusAsyncDevice, ModbusTcpDevice, ModbusTcpGatewayDevice):

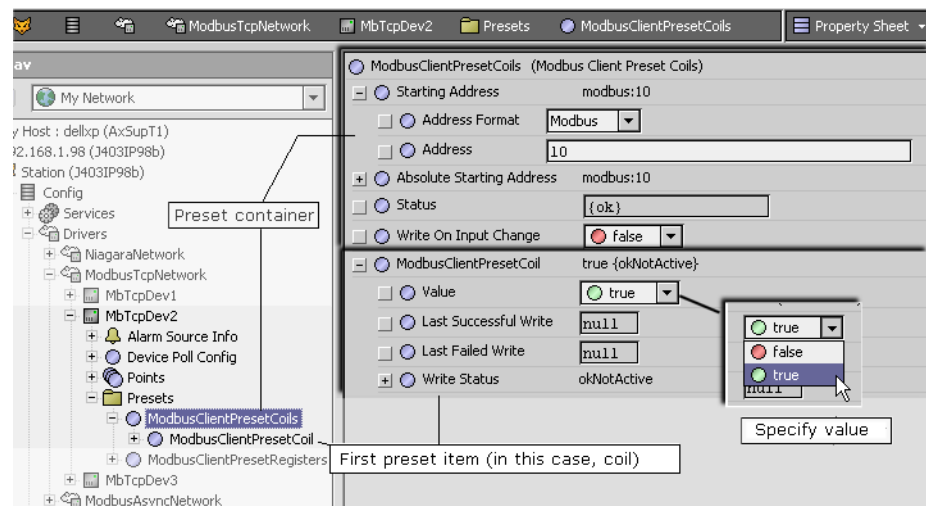
- **Modbus Client Preset Coils**
By default, this contains a single ModbusClientPresetCoil to write to one coil—you can add additional (consecutive) coils using a built-in action (see “Adding client presets”).
- **Modbus Client Preset Registers**
By default, this contains a single ModbusClientPresetRegister to write to one holding register—you can add additional (consecutive) registers using a built-in action (see “Adding client presets”).

Note: Preset components are not proxy points—you do not see them in any Modbus Client Point Manager view if you copy them under a client device’s **Points** extension. Therefore, it is recommended you locate them elsewhere under the device. Note also that you can simply copy the entire “Presets” folder from the palette if you want presets for both coils and holding registers, and then rename/edit as needed. Or, copy and paste already configured presets from another client Modbus device, if appropriate.

Modbus Client Preset Coils

One of two types of container **Modbus preset components**, the ModbusClientPresetCoils 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. Any additional child preset coil values are sequentially addressed relative to this (slot order). Note the “Absolute Address” is always used for actual addressing, which is typically the same as “Starting Address,” unless coils in the parent Modbus device are designated with a base address—see “Base Address properties” on page 4-14.

Figure 4-30 Starting Address in Modbus Client Preset Coils is for first child preset coil



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.

Properties of the ModbusClientPresetCoils container slot are as follows:

- **Starting Address**
Specifies the address of the first coil 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
Note if Modbus format, the leading “0s” are dropped for coil addresses, and the first coil is “1”.
 - Address — numerical address, expressed in the selected format.
 See “Modbus data addresses” on page 3-2 for general information.
- **Absolute Starting 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 writing the first coil’s preset value.
- **Status**
(Read only) Displays the status of the container slot—will be fault if a previous write to a child preset coil failed for some reason.
- **Write On Input Change**
Either false (default) or true. If set to true, any change to the boolean “value” of a child preset coil is immediately written to the target coil, in addition to the collective preset coil writes from invoking the “Write” action of the container. If false, preset coil writes occur only from the “Write” action.

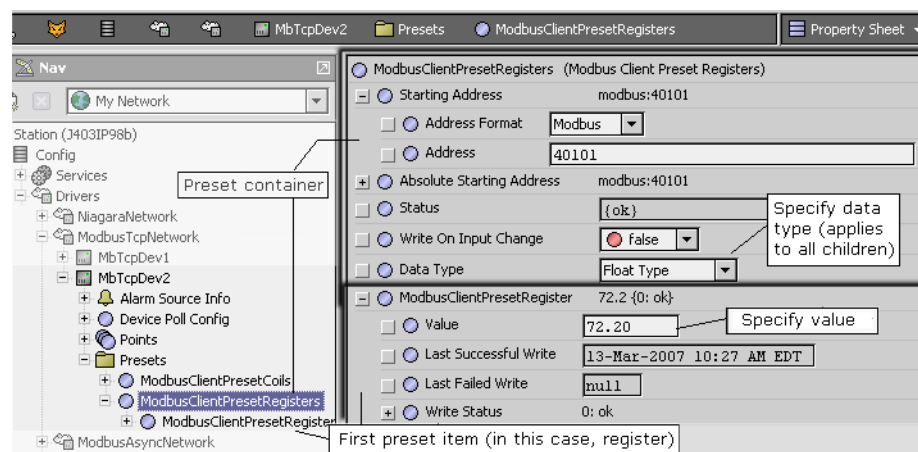
Actions for Modbus Client Preset Coils Two right-click actions on a [Modbus Client Preset Coils](#) container are as follows:

- **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 dialog. The component is added to the end of the existing slot order.

Modbus Client Preset Registers

One of two types of container [Modbus preset components](#), the ModbusClientPresetRegisters component contains one or more preset **holding register** values (ModbusClientPresetRegister components). In this container, you specify a “Starting Address” for the first (topmost) child preset register value. Any additional child preset register values are sequentially addressed relative to this (slot order). Note the “Absolute Address” is always used for actual addressing, which is typically the same as “Starting Address,” unless holding registers in the parent Modbus device are designated with a base address—see “[Base Address properties](#)” on page 4-14.

Figure 4-31 Starting Address in Modbus Client Preset Registers is for first child preset register



In this preset container you also 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.

Properties of the ModbusClientPresetRegisters container slot are as follows:

- **Starting Address**
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.

- See “Modbus data addresses” on page 3-2 for general information.
- **Absolute Starting 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 writing the first register’s preset value.
- **Status**
(Read only) Displays the status of the container slot—will be fault if a previous write to a child preset register failed for some reason.
- **Write On Input Change**
Either false (default) or true. If set to true, any change to the numeric “value” of a child preset register is immediately written to the target register, in addition to the collective preset register writes from invoking the “Write” action of the container. If false, preset register writes occur only from the “Write” action.
- **Data type**
Either Integer (default), Long, Float, or Signed Integer type. Specify to match the data type of the holding registers in the target Modbus device. Note *all* child preset registers must use this data type.

Actions for Modbus Client Preset Registers Two right-click actions on a [Modbus Client Preset Registers](#) container are as follows:

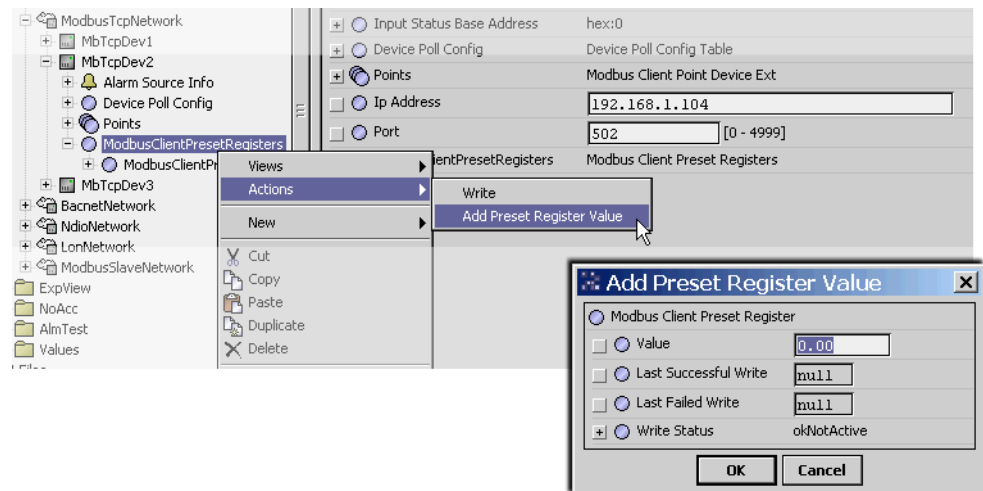
- **Write**
To write all the preset register values currently in child ModbusClientPresetCoil components.
- **Add Preset Register Value**
To add an additional child ModbusClientPresetRegister component, specifying its numerical value in the popup dialog. The component is added to the end of the existing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

Adding client presets

For any preset container ([Modbus Client Preset Coils](#), [Modbus Client Preset Registers](#)) you can add additional (consecutively addressed) child preset components using either of these two methods:

- Use the right-click “Add Preset Coil Value” or “Add Preset Register Value” *action* on either preset container type ([Figure 4-32](#)). This provides a dialog in which you specify another preset component, added under that preset type container.
- Manually, by duplicating/re-editing preset components (or by copying entries from the modbusAsync or modbusTcp palette and editing with a value as necessary).

Figure 4-32 Action “Add Preset Register Value” to create an additional consecutive register with preset value



By default, added client preset components are appended to the bottom of the slot order. If needed, you can right-click on a preset container and select “Reorder” to change the address order of the child presets, relative to the absolute address in the preset container.

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.*

About Modbus (client) file records

A Modbus Client String Record allows reading/writing Modbus file records (client 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, and reading occurs when the linkable “read” action is fired.

Note: Use of function codes 20 and 21 in Modbus devices is not typical, and so this object is expected to be infrequently used.

Configuration properties specify:

- File Number — From 0 to 65535.
- Starting Record Number — From 0 to 9999.
- Record Length — From 0 to 65535.
- Write On Input Change — Either false (default) or true.
- Padding — Either Pad with spaces (default) or Pad with nulls.

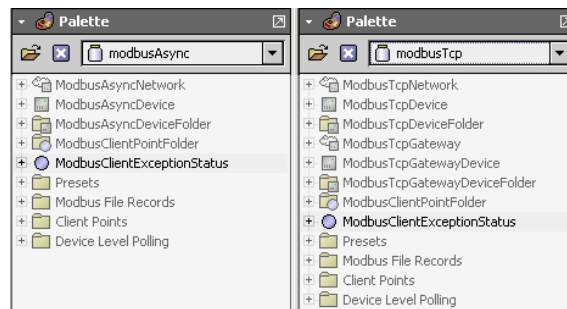
This component allows you to read and write file records in which data is converted to ASCII characters and displayed as a string.

To use, copy from the modbusAsync or modbusTcp palette and place anywhere under the Modbus client 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 Client Point Manager view).

About Modbus client exception status

Starting in AX-3.5 and later (or build 3.4.52 or higher), support was added for Modbus function code 07 (read Exception Status). A related component is now on the palette of Modbus client drivers (modbusAsync, modbusTcp): **ModbusClientExceptionStatus**, as shown in Figure 4-33.

Figure 4-33 ModbusClientExceptionStatus in the modbusAsync and modbusTcp palettes



Usage is “per device”, where 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.

Slots in this component include a “Bytes Returned” configuration property, where you specify whether the device uses 1 or 2 bytes for exception status, and also the poll frequency to be used. Read-only “Read Status” properties provide a numeric “Error Code” and “Error Description”, as well as timestamps of both the last successful and last failed read attempts.

Other output slots expose the exception status data as StatusBooleans on “Bit0” to “Bit15”, as well as a StatusNumeric “Out” slot.

Implementation of function code 07 (read Exception Status) *varies* among devices, but often is at least one returned byte that maps bits to various “unit status” information. For example, a vendor’s Modbus standby generator interface may respond to a function code 07 request with a byte of the following bitmapped data (as applied to any bit that is set):

- Bit 0 — Operative mode Off / Reset
- Bit 1 — Operative mode Manual
- Bit 2 — Operative mode Automatic
- Bit 3 — Operative mode Test
- Bit 4 — Error on
- Bit 5 — (not used)
- Bit 6 — (not used).
- Bit 7 — Global alarm on

Refer to the vendor’s documentation for a Modbus device to see if/how function code 07 is supported.

About Slave (server) types

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).

See the following sections:

- [About Modbus server devices](#)
- [About Modbus server proxy points](#)

About Modbus server devices

Modbus server (slave) devices include types ModbusSlaveDevice and ModbusTcpSlaveDevice. Both of these device components 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.

Note: *Device types are specific to a particular parent network type—for example, you cannot copy a ModbusSlaveDevice under a ModbusTcpSlaveNetwork, or a ModbusTcpSlaveDevice under 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.*

In addition to common device slots (see “Common device components” in the *Drivers Guide*), 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.

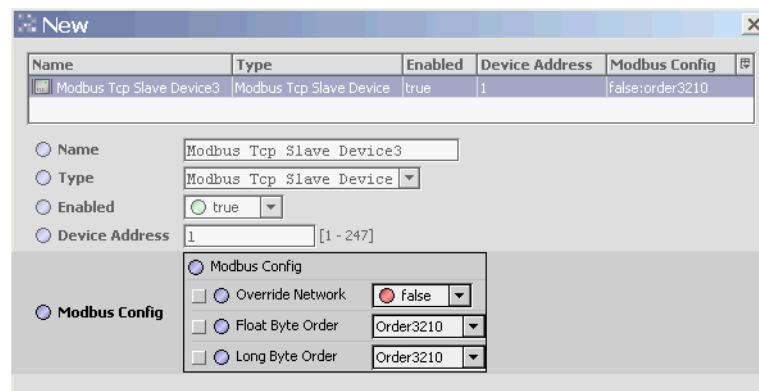
The following sections provide more details on these Modbus server device configuration:

- [Modbus Config \(server device level\)](#)
- [Modbus Register Range Tables](#)

Modbus Config (server device level)

Each Modbus server device has a “Modbus Config” container slot with 3 properties—you can access them on the device’s property sheet, as well as the New or Edit dialog for a device object when in the device manager view (of its parent network). [Figure 4-34](#) shows the properties in the New dialog for a device.

Figure 4-34 Modbus Config properties in New/Edit dialog when working in network’s device manager view



These properties allow you to “override” the network-level (global) equivalent settings for handling Modbus data from and to this virtual device, and are described as follows:

- **Override Network**
The default, false, means the network-level settings are used. Set this to true whenever you want the settings below to be used instead of the equivalent network-level values.
- **Float Byte Order**
Specifies the byte-order used to assemble or receive floating-point (32-bit) values in messages.
- **Long Byte Order**
Specifies the byte-order used to assemble or receive long integer (32-bit) values in messages.

Typically you leave these properties at defaults, particularly if only one Modbus server device—in which case you could adjust them (globally) at the network level.

Note: As a Modbus slave (server), by default Niagara supports Modbus function codes 15 and 16 (Force Multiple Coils, Preset Multiple Registers), so you do not see these selections like you do in a client Modbus device.

Modbus Register Range Tables

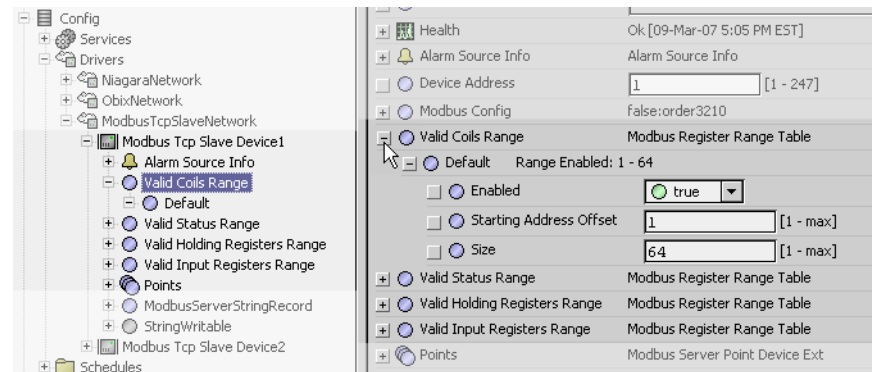
Each Modbus server device has four frozen container slots for setting “valid address ranges” for each of the four “virtual” data item types (coils, inputs, holding registers, input registers), as follows:

- Valid Coils Range
- Valid Status Range
- Valid Holding Registers Range
- Valid Input Registers Range

Queries received by the device must be for data items within the defined (and enabled) valid address ranges. Otherwise the station generates/sends an exception response back. 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.

You can see these range components under the device when expanded in the Nav tree, as well as in the device’s property sheet, as shown in [Figure 4-35](#).

Figure 4-35 Modbus Register Range Tables of Modbus server device



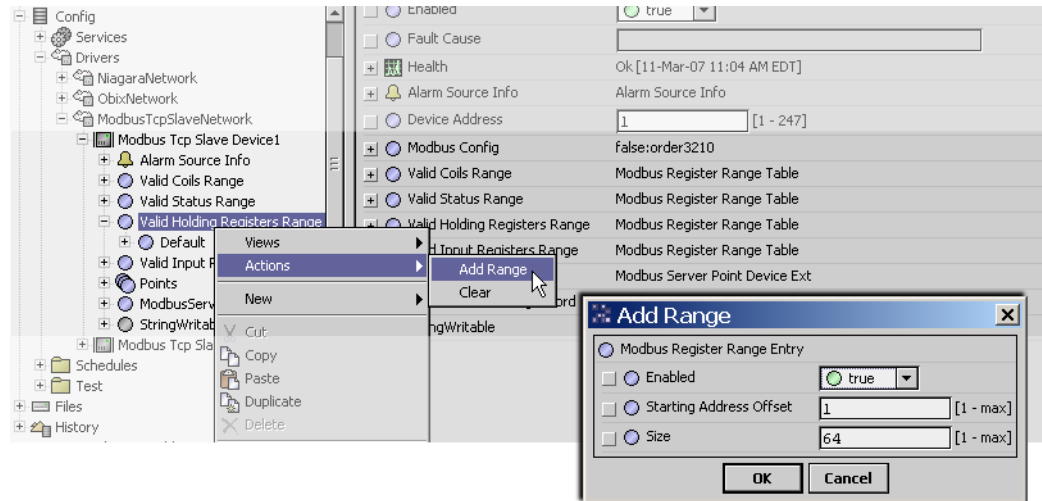
By default, as copied from the modbusSlave or modbusTcpSlave palette, a Modbus server device has a single “Default”-named Register Range Entry in each valid range container, with each having an enabled range of from 1 to 64, as shown in [Figure 4-35](#) for “Valid Coils Range.” The same default applies to the “Valid Holding Registers Range,” and so on.

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 this device to read discrete status (inputs) would yield an exception response. As another example, you could add multiple ranges for holding registers.

Configuring Valid Register Ranges You add additional register ranges using either of these two methods:

- Use the right-click “Add Range” **action** on any Modbus Register Range type ([Figure 4-36](#)). This provides a dialog in which you specify another [Modbus Register Range Entry](#) component, added under that range type container.
- Manually, by duplicating/re-editing Register Range Entry components (or by coping entries from the modbusSlave or modbusTcpSlave palette and editing as necessary).

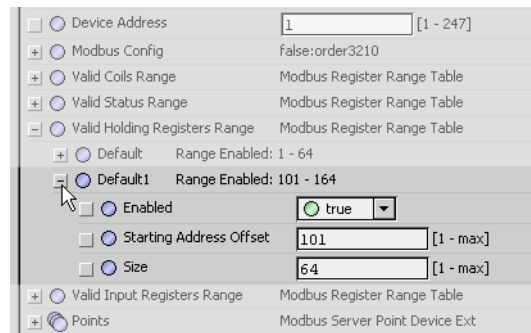
Figure 4-36 Action “Add Range” to create additional server register range entries



Each Modbus Register Range Table container also has a separate “Clear” action you can use to remove all existing Modbus Register Range Entry children.

Modbus Register Range Entry Figure 4-37 shows a Modbus Register Range Entry child of one of the [Modbus Register Range Tables](#) expanded, and its properties that were specified.

Figure 4-37 Expanded Modbus Register Range Entry in Modbus server device



Properties of a Modbus Register Range Entry are described as follows:

- **Enabled** — By default, is true. If set to false, any associated [Modbus server proxy points](#) will have a fault status, and any queries received to this address range result in an exception response
- **Starting Address Offset** — The first register in the specified range.
- **Size** — The number of registers in the range.

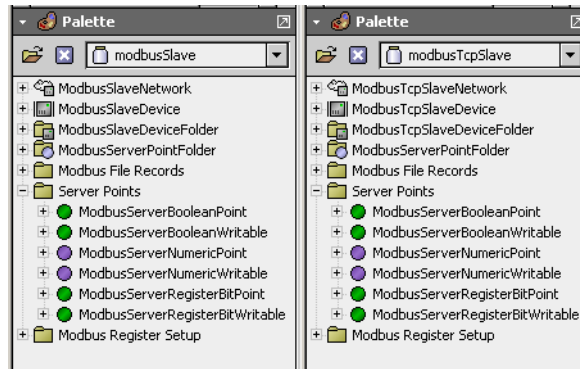
For example, if a holding register range is set to a starting address of 250 with a size of 75, it will have an effective Modbus address range of 40250 to 40325.

Note: *If using multiple ranges under any of the [Modbus Register Range Tables](#), be careful not to “overlap” any range entries. In other words, any specific register address should be in only **one** range entry.*

About Modbus server proxy points

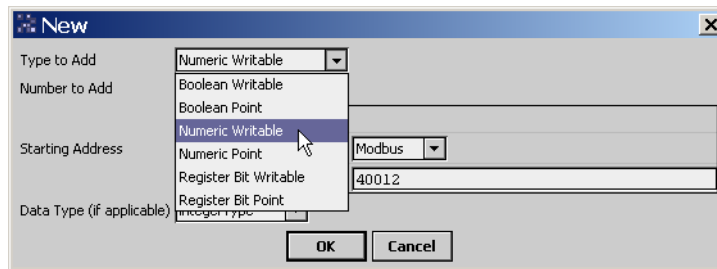
Modbus server proxy points are similar to other driver’s proxy points. See “About proxy points” in the *Drivers Guide* for general details. Note that the **same collection** of server proxy points is used in devices under a ModbusSlave and ModbusTcpSlave network—you can find them in the “Server Points” folder in either palette type (modbusSlave or modbusTcpSlave), as shown in [Figure 4-27](#).

Figure 4-38 Server Points folder is same in modbusSlave palette and modbusTcpSlave palette



Although sometimes you may need to copy components from the palette, note that the same selection of server point types is available in the **New** dialog, when adding points in the Modbus Server Point Manager view of a device (of its Points extension), as shown in [Figure 4-39](#).

Figure 4-39 New dialog from Add in Modbus Server Point Manager provides server point selection



Typically, this is the quickest way to add Modbus server proxy points, because you can specify a number of points if consecutively addressed. See “[Consecutive address usage \(NiagaraAX\)](#)” on page 3-4.

The following sections provide additional details on server Modbus proxy points:

- [Types of Modbus server proxy points](#)
- [Modbus server point ProxyExt properties](#)

Types of Modbus server proxy points

Modbus server proxy points represent “virtual” Modbus data items in the parent Modbus server device. You can select from the following Modbus proxy point types:

- [Boolean Writable](#)
 To read/write a “virtual” Modbus coil or input.
Note: Generally, it is unwise to expose any coil as BooleanWritable if the Modbus master may also write to this same item—otherwise “write contention” issues may result.
- [Boolean Point](#)
 To read a “virtual” Modbus coil that may be written by the Modbus master.
- [Numeric Writable](#)
 To read/write a “virtual” Modbus holding register value or input register value.
 Note you must specify the Data Type, as either integer, long, float, or signed integer.
Note: 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](#)
 To read 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](#)
 To read/write a specific bit in a “virtual” Modbus holding register or input register (select Bit Num-

ber 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](#)
To read a specific bit in a “virtual” Modbus holding register (select Bit Number in setup) that may be written by the Modbus master.
- [String Point](#)
To read 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.

Modbus server point ProxyExt properties

Apart from the standard “core” proxy extension properties (see “ProxyExt properties” in the *Drivers Guide*), these ProxyExt properties have special importance in Modbus client proxy points:

- **Fault Cause**
(Read only) If the point is in fault due to setup error (not within a [Modbus Register Range Entry](#) of the parent server device) the Fault Cause includes the generated exception string. For example: “Read fault: Exception during read (com.tridium.modbusCore.ModbusException: DATA_NOT_AVAILABLE)”. See “[Exception codes](#)” on page 3-10 for more details.
- **Read Value**
(Read only) Shows last polled value as well as the state, for example “71 {ok}” or “false {ok}”.
- **Write Value**
(Read Only) Shows the last written value, as well as the state and priority level, for example “70 {overridden} @8”.
- **Data Address**
Specifies the address of the polled data item using a combination of:
 - Address Format — either Modbus (default), Hex, or Decimal
 - Address — numerical address, expressed in the selected format.See “[Modbus data addresses](#)” on page 3-2 for general information.
For example, the following are all equivalent addresses:
 - Modbus, 40012
 - Hex, 0B
 - Decimal, 11

Note: If using Hex or Decimal format, for most points you need to specify the “Reg Type” property, to clarify whether holding register or input register. For related details, see “[Data address format in NiagaraAX](#)” on page 3-5.

Depending on the *type* of Modbus proxy point, one or more *additional* properties are used in its ProxyExt to clarify or confirm the data item needed, as follows:

- [ModbusServerBooleanPoint](#)
- [ModbusServerBooleanWritable](#)
- [ModbusServerNumericPoint](#)
- [ModbusServerNumericWritable](#)
- [ModbusServerRegisterBitPoint](#)
- [ModbusServerRegisterBitWritable](#)

ModbusServerBooleanPoint Or “Boolean Point”, has the following in addition to other [Modbus server point ProxyExt properties](#):

- **Status Type**
Coil or Input type to read, however Coil is the only valid selection (master cannot write to Modbus inputs). Selection necessary only if the “Data Address” format used is Hex or Decimal. (The Modbus address format, if used, automatically sets this property value).

ModbusServerBooleanWritable Or “Boolean Writable”, has the following in addition to other [Modbus server point ProxyExt properties](#):

- **Status Type**
Coil or Input. Specifies whether the point is read from/written to a coil status or input status type. Selections only apply if the “Data Address” format used is Hex or Decimal. (The Modbus address format, if used, automatically sets this property value).

ModbusServerNumericPoint Or “Numeric Point”, has the following in addition to other [Modbus server point ProxyExt](#) properties:

- **Reg Type**
Holding or Input, however Holding is the only valid selection (master cannot write to Modbus input registers). Selection is necessary only if the “Data Address” format used is Hex or Decimal. (The Modbus address format, if used, automatically sets this property value).
- **Data Type**
Specifies the data type used by the associated data point. Integer and signed integer are 16-bit (single register) data types; long and float are 32-bit types (with the starting address specified in “Data Address”). Values for long and float selections are based upon the network's byte-order config setup (or may be overridden at the device-level).

ModbusServerNumericWritable Or “Numeric Writable”, has the following in addition to other [Modbus server point ProxyExt](#) properties:

- **Reg Type**
Holding or Input, to specify whether the numeric value is read from/written to a holding register or input register.
- **Data Type**
Specifies the data type used by the associated data point. Integer and signed integer are 16-bit (single register) data types; long and float are 32-bit types (with the starting address specified in “Data Address”). Values for long and float selections are based upon the network's byte-order config setup (or may be overridden at the device-level).

ModbusServerRegisterBitPoint Or “Register Bit Point”, has the following in addition to other [Modbus server point ProxyExt](#) properties:

- **Reg Type**
Holding or Input, however Holding is the only valid selection (master cannot write to Modbus input registers). Selections only apply if the “Data Address” format used is Hex or Decimal. (The Modbus address format, if used, automatically sets this property value).
- **Bit Number**
Specifies the bit (numbered 0 - 15, least significant bit first) to read from the specified (16-bit) Modbus register. For example, if the specified register value was “0000000000001000”, setting the “Bit Number” to 3 would read a “1” (True).

ModbusServerRegisterBitWritable Or “Register Bit Writable”, has the following in addition to other [Modbus server point ProxyExt](#) properties:

- **Reg Type**
Holding or Input, to specify whether the boolean value is read from/written to a bit in either a holding register or input register.
- **Bit Number**
Specifies the bit (numbered 0 - 15, least significant bit first) to write to the specified (16-bit) Modbus register. For example, if the specified register value was “0000000000000000”, setting the “Bit Number” to 3 and writing a True (“1”) would cause the specified register value to become “0000000000001000”.

About Modbus (server) file records

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.

Note: *Use of function codes 20 and 21 in Modbus devices is not typical, and so this object is expected to be infrequently used.*

Configuration properties specify:

- File Number — From 0 to 65535.
- Starting Record Number — From 0 to 9999.
- Record Length — From 0 to 65535.
- Write On Input Change — Either false (default) or true.
- Padding — Either Pad with spaces (default) or Pad with nulls.

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).

CHAPTER 5

Modbus Plugin Guides

There are many ways to view plugins (*views*). One way is directly in the tree. In addition, you can right-click on an item and select one of its views. Plugins provide views of components. You can access documentation on a Plugin by selecting **Help > On View (F1)** from the menu, or by pressing F1 while the Plugin is selected.

These Plugin Guides provide summary information on Modbus views, listed alphabetically by module.

Plugin Guides Summary

The following Modbus plugins are available, by module:

- [modbusAsync](#)
- [modbusCore](#)
- [modbusSlave](#)
- [modbusTcp](#)
- [modbusTcpSlave](#)

modbusAsync plugins

The following views apply to the Modbus Async driver:

- [ModbusAsyncDeviceManager](#)

modbusAsync-ModbusAsyncDeviceManager

Use the [ModbusAsyncDeviceManager](#) to create, edit, and view Modbus Async Devices. The [ModbusAsyncDeviceManager](#) is a view on the [ModbusAsyncNetwork](#). To view, double-click the [ModbusAsyncNetwork](#), or right-click and select **Views > Modbus Async Device Manager**.

For general information, see “About the Device Manager” in the *Drivers Guide*. See “[Modbus Async Device Manager notes](#)” on page 4-3 for additional details.

modbusCore plugins

The following views are found in multiple types of Modbus networks:

- [ModbusClientPointManager](#)
- [ModbusServerPointManager](#)

modbusCore-ModbusClientPointManager

Use the [ModbusClientPointManager](#) to create, edit, and view 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. To view, double-click any of those components.

For general information, see the section “About the Point Manager” in the *Drivers Guide*. See “[Modbus Client Point Manager notes](#)” on page 4-16 for additional details.

modbusCore-ModbusServerPointManager

Use the [ModbusServerPointManager](#) to create, edit, and view proxy points under a server Modbus device. It is the default view on the Points container of a [ModbusSlaveDevice](#) and [ModbusTcpSlaveDevice](#), as well as on any points folder under those Points containers. To view, double-click any of those components.

For general information, see the section “About the Point Manager” in the *Drivers Guide*.

modbusSlave plugins

The following views apply to the Modbus Slave driver:

- [ModbusSlaveDeviceManager](#)

modbusSlave-ModbusSlaveDeviceManager

Use the ModbusSlaveDeviceManager to create, edit, and view Modbus Slave Devices. The ModbusSlaveDeviceManager is a view on the [ModbusSlaveNetwork](#). To view, double-click a [ModbusSlaveNetwork](#), or right-click and select **Views > Modbus Slave Device Manager**.

For general information, see “About the Device Manager” in the *Drivers Guide*.

modbusTcp plugins

The following views apply to the Modbus TCP drivers:

- [ModbusTcpDeviceManager](#)
- [ModbusTcpGatewayDeviceManager](#)

modbusTcp-ModbusTcpDeviceManager

Use ModbusTcpDeviceManager to create, edit, and view Modbus Tcp Devices. The ModbusTcpDeviceManager is a view on the [ModbusTcpNetwork](#). To view, double-click a [ModbusTcpNetwork](#), or right-click and select **Views > Modbus Tcp Device Manager**.

For general information, see “About the Device Manager” in the *Drivers Guide*. See “[Modbus Tcp Device Manager notes](#)” on page 4-5 for additional details.

modbusTcp-ModbusTcpGatewayDeviceManager

Use ModbusTcpGatewayDeviceManager to create, edit, and view Modbus Tcp Gateway Devices. The ModbusTcpGatewayDeviceManager is a view on the [ModbusTcpGateway](#). To view, double-click a [ModbusTcpGateway](#), or right-click and select **Views > Modbus Tcp Gateway Device Manager**.

For general information, see “About the Device Manager” in the *Drivers Guide*. See “[Modbus Tcp Gateway Device Manager notes](#)” on page 4-8 for additional details.

modbusTcpSlave plugins

The following views apply to the Modbus TCP Slave driver:

- [ModbusTcpSlaveDeviceManager](#)

modbusTcpSlave-ModbusTcpSlaveDeviceManager

Use ModbusTcpSlaveDeviceManager to create, edit, and view Modbus Tcp Slave Devices. The ModbusTcpSlaveDeviceManager is a view on the [ModbusTcpSlaveNetwork](#). To view, double-click a [ModbusTcpSlaveNetwork](#), or right-click and select **Views > Modbus Tcp Slave Device Manager**.

For general information, see “About the Device Manager” in the *Drivers Guide*.

CHAPTER 6

Modbus Component Guides

These Component Guides provide summary information on Modbus components, listed alphabetically by module.

Component Reference Summary

Summary information is provided on components in the following modules:


- [modbusAsync](#)
- [modbusCore](#)
- [modbusSlave](#)
- [modbusTcp](#)
- [modbusTcpSlave](#)

modbusAsync components

In addition to “client” components in the [modbusCore](#) module, the following components apply to the Modbus Async driver:

- [ModbusAsyncDevice](#)
- [ModbusAsyncDeviceFolder](#)
- [ModbusAsyncNetwork](#)


modbusAsync-ModbusAsyncDevice

 The [ModbusAsyncDevice](#) represents a Modbus serial (async) device under a [ModbusAsyncNetwork](#), for 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 configuration, including slots to specify the base address for its Modbus data items (holding registers, input registers, inputs, coils), plus a [DevicePollConfigTable](#) for device polling.

Its Points extension ([ModbusClientPointDeviceExt](#)) contains Modbus proxy points with client proxy extensions ([ModbusClientBooleanProxyExt](#), [ModbusClientNumericProxyExt](#), [ModbusClientRegisterBitProxyExt](#)), used to read and write data to the defined data items. The device can also contain one or more [ModbusClientStringRecord](#) components.


For more details, see “[About Modbus client devices](#)” on page 4-12.

modbusAsync-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 [ModbusAsyncDeviceManager](#) view of the network. Each device folder has its own device manager view. The [ModbusAsyncDeviceFolder](#) is also available in the modbusAsync palette.

modbusAsync-ModbusAsyncNetwork

 [ModbusAsyncNetwork](#) 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 has other standard network components, including a Serial Port Config container ([SerialHelper](#)) to specify serial settings used by the JACE for communications.

See “[About Modbus Async networks](#)” on page 4-1 for additional details.


modbusCore Components

Core Modbus components are common to *multiple* Modbus network types, either the 3 “client” types ([ModbusAsyncNetwork](#), [ModbusTcpNetwork](#), [ModbusTcpGateway](#)) or the 2 “server” types ([ModbusSlaveNetwork](#), [ModbusTcpSlaveNetwork](#)), and include the following:


- [DevicePollConfigEntry](#)
- [DevicePollConfigTable](#)
- [ModbusClientBooleanProxyExt](#)
- [ModbusClientEnumBitsProxyExt](#) (build 3.5.26 or higher, AX-3.6 or later)
- [ModbusClientExceptionStatus](#) (build 3.4.52 or higher, or AX-3.5 and later)
- [ModbusClientNumericBitsProxyExt](#) (build 3.5.26 or higher, AX-3.6 or later)
- [ModbusClientNumericProxyExt](#)
- [ModbusClientPointDeviceExt](#)
- [ModbusClientPointFolder](#)
- [ModbusClientPresetCoil](#)
- [ModbusClientPresetCoils](#)
- [ModbusClientPresetRegister](#)
- [ModbusClientPresetRegisters](#)
- [ModbusClientRegisterBitProxyExt](#)
- [ModbusClientStringProxyExt](#)
- [ModbusClientStringRecord](#)
- [ModbusRegisterRangeEntry](#)
- [ModbusRegisterRangeTable](#)
- [ModbusServerBooleanProxyExt](#)
- [ModbusServerNumericProxyExt](#)
- [ModbusServerPointDeviceExt](#)
- [ModbusServerPointFolder](#)
- [ModbusServerRegisterBitProxyExt](#)
- [ModbusServerStringRecord](#)

Note: There is no “*modbusCore palette*”—core Modbus components are in the various other Modbus palettes.

modbusCore-DevicePollConfigEntry

 DevicePollConfigEntry is used to configure 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* Modbus device. Properties specify the starting Modbus address and number of points to poll. For more details, see “[Device Poll Config Entry](#)” on page 4-16.


modbusCore-DevicePollConfigTable

 This is a frozen slot under a *client* Modbus device ([ModbusAsyncDevice](#), [ModbusTcpDevice](#), or [ModbusTcpGatewayDevice](#)). It can contain a table of [DevicePollConfigEntry](#)s for specifying device-level polling of points within the device. The Device Poll Config table has two available *actions*:

- Learn Optimum Device Poll Config — To automatically create child DevicePollConfigEntry components based upon the current collection of Modbus proxy points.
- Clear — To remove all existing child DevicePollConfigEntry components.


For more details, see “[Device Poll Config](#)” on page 4-15.

modbusCore-ModbusClientBooleanProxyExt

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


For more details, see “[Types of Modbus client proxy points](#)” on page 4-18.

modbusCore-ModbusClientEnumBitsProxyExt

 (build 3.5.26 or higher, or AX-3.6 and later) 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.

For more details, see “[Types of Modbus client proxy points](#)” on page 4-18.


modbusCore-ModbusClientExceptionStatus

 (build 3.4.52 or higher, or AX-3.5 and later) ModbusClientExceptionStatus is a component used to read and expose Modbus function code 07 (Exception Status) data from a Modbus device. 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](#).

Slots in this component include a “Bytes Returned” configuration property, where you specify whether the device uses 1 or 2 bytes for exception status, and also the poll frequency to be used. Read-only “Read Status” properties provide a numeric “Error Code” and “Error Description”, as well as timestamps of both the last successful and last failed read attempts. Other output slots expose the exception status data as StatusBooleans on “Bit0” to “Bit15”, as well as a StatusNumeric “Out” slot.


For more details, see “[About Modbus client exception status](#)” on page 4-25.

modbusCore-ModbusClientNumericBitsProxyExt

 (build 3.5.26 or higher, or AX-3.6 and later) 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 specified 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.


For more details, see “[Types of Modbus client proxy points](#)” on page 4-18.

modbusCore-ModbusClientNumericProxyExt

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


For more details, see “[Types of Modbus client proxy points](#)” on page 4-18.

modbusCore-ModbusClientPointDeviceExt


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

For more details, see “[Types of Modbus client proxy points](#)” on page 4-18.


modbusCore-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 [ModbusClientPointManager](#). Each points folder also has its own Point Manager view.

modbusCore-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.

modbusCore-ModbusClientPresetCoils

 ModbusClientPresetCoils is a container used for writing preset Modbus *coil* (boolean) data values to a *client* Modbus device. As copied from the palette, a single child [ModbusClientPresetCoil](#) entry exists in this component, where you can enter a preset value (false or true) for that coil. Also, you can add additional (consecutive) preset coil entries, each with its own value, by using the “Add Preset Coil Value” action (or by duplicating or by copying from the palette).

ModbusClientPresetCoils configuration requires specifying the Starting Address of the first coil.

Writing occurs when the linkable “write” action is fired. Values specified in the dynamic children are written to the device using the specified starting (absolute) address, and the slot order of the children determines what (consecutive) address is assigned to their values.

As needed, you can copy these components from the modbusAsync or modbusTcp palette into a [ModbusAsyncDevice](#), [ModbusTcpDevice](#), or [ModbusTcpGatewayDevice](#).

Note: Presets (registers and coils) provide write-access only, and are not proxy points—they do not produce polling activity. If copied under the Points container of the client Modbus device, they will not be visible in the Point Manager. You may wish to add a “Presets” container under the device to keep any preset objects. For more details, see “Modbus Client Preset Coils” on page 4-22.

modbusCore-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.

modbusCore-ModbusClientPresetRegisters

○ ModbusClientPresetRegisters is a container used for writing preset Modbus *holding register* data values to a *client* Modbus device. As copied from the palette, a single child [ModbusClientPresetRegister](#) entry exists in this component, where you can enter a preset value for that register. Also, you can add additional (consecutive) preset register entries, each with its own value, by using the “Add Preset Register Value” action (or by duplicating or by copying from the palette).

ModbusClientPresetRegisters configuration requires specifying the Starting Address of the first register, and also the Data Type (Integer, Float, Long, Signed Integer). All child preset register components must use this data type.

Writing occurs when the linkable “Write” action is fired. Values specified in the dynamic children are written to the device using the specified starting (absolute) address, and the slot order of the children determines what (consecutive) address is assigned to their values.

As needed, you can copy these components from the modbusAsync or modbusTcp palette into a [ModbusAsyncDevice](#), [ModbusTcpDevice](#), or [ModbusTcpGatewayDevice](#).

Note: Presets (registers and coils) provide write-access only, and are not proxy points—they do not produce polling activity. If copied under the Points container of the client Modbus device, they will not be visible in the Point Manager. You may wish to add a “Presets” container under the device to keep any preset objects. For more details, see “Modbus Client Preset Coils” on page 4-22.

modbusCore-ModbusClientRegisterBitProxyExt

■ 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. For more details, see “Types of Modbus client proxy points” on page 4-18.

modbusCore-ModbusClientStringProxyExt

■ 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.

For more details, see “Types of Modbus client proxy points” on page 4-18.

modbusCore-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](#). For more details, see “About Modbus (client) file records” on page 4-25.

modbusCore-ModbusRegisterRangeEntry

○ ModbusRegisterRangeEntry is used to configure 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.


If needed, more than one register range entry can be added under any register table. For more details, see “Modbus Register Range Entry” on page 4-28.

modbusCore-ModbusRegisterRangeTable

 ModbusRegisterRangeTable is used to define the data constructs of a *server* Modbus device, where each device ([ModbusSlaveDevice](#) or [ModbusTcpSlaveDevice](#)) has *four* frozen instances of these tables: one each for valid coils, status (inputs), holding registers, and input registers.


Each register range table can have one or more child [ModbusRegisterRangeEntry](#) components, which define the address offset and range of the data item. For more details, see “[Modbus Register Range Tables](#)” on page 4-27.

modbusCore-ModbusServerBooleanProxyExt

 This is the proxy extension for either a [ModbusServerBooleanPoint](#) ([BooleanPoint](#)) or [ModbusServerBooleanWritable](#) ([BooleanWritable](#)) in a server Modbus device. The server [BooleanPoint](#) acts as an “input,” where coil values are written by the master. The server [BooleanWritable](#) acts as an “output,” where station values can be written as either coil or input (status) values.


For more details, see “[About Modbus server proxy points](#)” on page 4-29.

modbusCore-ModbusServerNumericProxyExt

 This is the proxy extension for either a [ModbusServerNumericPoint](#) ([NumericPoint](#)) or [ModbusServerNumericWritable](#) ([NumericWritable](#)) in a server Modbus device. The server [NumericPoint](#) acts as an “input,” where holding register values are written by the master. The server [NumericWritable](#) acts as an “output,” where station values can be written as either input register or holding register values.


For more details, see “[About Modbus server proxy points](#)” on page 4-29.

modbusCore-ModbusServerPointDeviceExt


 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 [ModbusServerPointManager](#).

For more details, see “[About Modbus server proxy points](#)” on page 4-29.

modbusCore-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 [ModbusServerPointManager](#). Each points folder also has its own Point Manager view.

modbusCore-ModbusServerRegisterBitProxyExt

 This is the proxy extension for either a [ModbusServerRegisterBitPoint](#) ([BooleanPoint](#)) or [ModbusServerRegisterBitWritable](#) ([BooleanWritable](#)) in a server Modbus device. The server [BooleanPoint](#) acts as an “input,” where an individual bit in a holding register can be written by the master. The server [BooleanWritable](#) acts as an “output,” where a station boolean value can be written as an individual bit in either an input register or a holding register.

For more details, see “[About Modbus server proxy points](#)” on page 4-29.

modbusCore-ModbusServerStringRecord

 [ModbusServerStringRecord](#) is a component for reading/writing Modbus file records (server 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.


For more details, see “[About Modbus \(server\) file records](#)” on page 4-31.

modbusSlave components

In addition to “server” components in the [modbusCore](#) module, the following components apply to the Modbus Slave driver:

- [ModbusSlaveDevice](#)
- [ModbusSlaveDeviceFolder](#)
- [ModbusSlaveNetwork](#)


modbusSlave-ModbusSlaveDevice

 The [ModbusSlaveDevice](#) represents a “virtual” Modbus device to serve data to a Modbus master over a *serial* 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](#), [ModbusServerRegister-BitProxyExt](#)), used to read and write data to the defined data items. The device can also contain one or more [ModbusServerStringRecord](#) components.


For more details, see “[About Modbus server devices](#)” on page 4-26.

modbusSlave-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 [ModbusSlaveDeviceManager](#) view of the network. Each device folder has its own device manager view. The [ModbusSlaveDeviceFolder](#) is also available in the modbusSlave palette.

modbusSlave-ModbusSlaveNetwork

 [ModbusSlaveNetwork](#) is the base container for one or more [ModbusSlaveDevice](#) components. This network component specifies the Modbus data mode (RTU or ASCII) used by the network, and has other standard network components, including a Serial Port Config container ([SerialHelper](#)) to specify serial settings used by the JACE for communications.


See “[About Modbus Slave networks](#)” on page 4-8 for additional details.

modbusTcp components

In addition to “client” components in the [modbusCore](#) module, the following components apply to the Modbus TCP driver:

- [ModbusTcpDevice](#)
- [ModbusTcpDeviceFolder](#)
- [ModbusTcpGateway](#)
- [ModbusTcpGatewayDevice](#)
- [ModbusTcpGatewayDeviceFolder](#)
- [ModbusTcpNetwork](#)


modbusTcp-ModbusTcpDevice

 The [ModbusTcpDevice](#) represents a Modbus TCP device under a [ModbusTcpNetwork](#), for 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, and other configuration including slots to specify the base address for its Modbus data items (holding registers, input registers, inputs, coils), plus a [DevicePollConfigTable](#) for device polling.

Its Points extension ([ModbusClientPointDeviceExt](#)) contains Modbus proxy points with client proxy extensions ([ModbusClientBooleanProxyExt](#), [ModbusClientNumericProxyExt](#), [ModbusClientRegister-BitProxyExt](#)), used to read and write data to the defined data items. The device can also contain one or more [ModbusClientStringRecord](#) components.


For more details, see “[About Modbus client devices](#)” on page 4-12.

modbusTcp-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 [ModbusTcpDeviceManager](#) view of the network. Each device folder has its own device manager view. The [ModbusTcpDeviceFolder](#) is also available in the modbusTcp palette.

modbusTcp-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](#).

In addition to standard network components, other Modbus settings are specified in this component. Its primary view is the [ModbusTcpGatewayDeviceManager](#), used to add and manage devices. See “[About Modbus TCP Gateway networks](#)” on page 4-6 for additional details.


modbusTcp-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). In addition to the typical device components, it specifies its Modbus address plus other settings, including slots to specify the base address for its Modbus data items (holding registers, input registers, inputs, coils), and a [DevicePollConfigTable](#) for device polling.

Its Points extension ([ModbusClientPointDeviceExt](#)) contains Modbus proxy points with client proxy extensions ([ModbusClientBooleanProxyExt](#), [ModbusClientNumericProxyExt](#), [ModbusClientRegisterBitProxyExt](#)), used to read and write data to the defined data items. The device can also contain one or more [ModbusClientStringRecord](#) components.


For more details, see “[About Modbus client devices](#)” on page 4-12.

modbusTcp-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 [ModbusTcpGatewayDeviceManager](#) view of the network. Each device folder has its own device manager view. The [ModbusTcpGatewayDeviceFolder](#) is also available in the modbusTcp palette.

modbusTcp-ModbusTcpNetwork

 ModbusTcpNetwork is the base container for one or more [ModbusTcpDevice](#) components. This network component specifies certain Modbus settings used by the network, and has other standard network components. Its primary view is the [ModbusTcpDeviceManager](#), used to add and manage devices.


See “[About Modbus TCP networks](#)” on page 4-4 for additional details.

modbusTcpSlave components

In addition to “server” components in the [modbusCore](#) module, the following components apply to the Modbus TCP Slave driver:

- [ModbusTcpSlaveDevice](#)
- [ModbusTcpSlaveDeviceFolder](#)
- [ModbusTcpSlaveNetwork](#)

modbusTcpSlave-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.


For more details, see “[About Modbus server devices](#)” on page 4-26.

modbusTcpSlave-ModbusTcpSlaveDeviceFolder

 This is the ModbusTcpSlave implementation of a folder under a [ModbusTcpSlaveNetwork](#). You can use these folders to organize [ModbusTcpSlaveDevices](#) in the network.

Typically, you add such folders using the **New Folder** button in the [ModbusTcpSlaveDeviceManager](#) view of the network. Each device folder has its own device manager view. The [ModbusTcpSlaveDeviceFolder](#) is also available in the modbusTcpSlave palette.

modbusTcpSlave-ModbusTcpSlaveNetwork

 ModbusTcpSlaveNetwork is the base container for one or more [ModbusTcpSlaveDevice](#) components. This network component specifies the TCP connection settings used to expose the virtual devices to Modbus, and assumes the IP address of the Niagara station.

For more details, see “[About Modbus TCP Slave networks](#)” on page 4-10.

