Advanced Modbus Module User Manual

Modbus Client and Server Driver Suite for Ignition

Version 1.2.4-v8.3

November 4, 2026 17:19z

The Advanced Modbus™ Communications Module for Ignition™ implements either end of a Modbus TCP or Modbus RTU communication channel. Modbus RTU is supported on local Serial Ports and on remote Serial Ports via raw TCP connections.



Table of Contents

Overview....3
Modbus OPC Addressing....4
  Supported Address Formats....4
Advanced Modbus Server....5
  Supported Function Codes....5
  Data Consistency....5
  Modbus TCP....6
  Modbus RTU (Local and Remote Serial)....6
  Device Settings....6
    General....6
    Communications....6
    Behavior....7
  Device Units Configuration....7
    Import/Export....8
    Server Unit Detail Configuration CSV Columns....8
    Automated Device Creation....8
Advanced Modbus Client....10
  Supported Function Codes....10
  Device Settings....10
    General....11
    Communications....11
    Modbus TCP....11
    Modbus RTU Serial....11
    Modbus RTU via TCP....12
    Behavior....12
  Device Units Configuration....13
    Import/Export....13
    Client Unit Detail Configuration CSV Columns....13
    Read Request Optimization....14
    Write Request Options....14
    Automated Device Creation....14
  Scripting Arbitrary Modbus Requests....15
Moxa NPort Application Notes....16
Linux Serial Port Application Notes....17
  Low Latency Mode....17
  Serial Device Access Privileges....17
  RTU Frame Timing Adjustment....17

Overview

The Advanced Modbus Module enables Inductive Automation's Ignition platform to communicate with a large variety of devices that offer support for the venerable Modbus Application Protocol. The core addressing syntax is the same as for Ignition's native Modbus driver. This module provides several additional features, both functional and syntactic, as follows:

- *Serve* data to external Modbus clients via both TCP and RTU protocols. The same Modbus function codes a client connection can issue are fully implemented in this server driver, with multiple virtual slaves emulated in a single driver instance. Multiple simultaneous client connections to these emulated slaves are supported, and may be any connection type (listening ports).
- *Browse* the full range of bit and word addresses in driver instances, per slave unit, out of the box. By default, slave unit #1 is assumed to exist and contain a full complement of coils, discrete inputs, analog inputs, and holding registers (64k of each). Addresses available in different slave units of a connection are *individually configurable by slave unit number, in both client and server drivers.*
- *Browse* all allowed data type transformations for configured word addresses as a subtree under the "plain" word address. Configure transformations like word swapping for large integers and floats, and byte swapping for character strings, *per slave unit*.
- Read and write word addresses in *memory area #6*, "File Records", using Modbus function codes 20 and 21 (0x16 and 0x17), with a configurable memory size *per slave unit*. Use prefix "XR" for file record registers with all the normal word-register data type suffixes.
- Treat multiple sequential discrete input bits or coils (C and DI prefixes) as unsigned integer bit fields, up to 63 bits long. Treat multiple sequential bits of single word registers (IR, HR, and XR prefixes) as unsigned integer bit fields (up to 15 bits).
- Where protocol compatibility tweaks are required, like control over gap spanning and other optimizations, the settings are configurable *per slave unit*. Where memory areas are configured with specific address ranges, the configured gaps are *automatically* excluded from spanning, allowing users to leave that optimization in place for other addresses.
- For client connections, script Modbus requests with arbitrary function and payload, and receive the reply.

Two features of Ignition's native driver have been omitted from this Advanced Modbus Driver:

- User-defined browseable address mappings, and
- One-based addressing.

The address mapping feature, which is the only browseable part of the native driver, conflicts with this driver's new generic browsing functionality. One-based addressing was omitted to ensure the configurable address ranges were always unambiguous. As a convenience/reminder, the one-based modbus address with classic prefix (0, 1, 3, 4, 6) is shown in this module's browse text for each "plain" memory area.

Finally, this module defaults to unit #1 instead of unit #0 when omitted from an OPC item address. The Modbus Specification declares the zero unit address to be the broadcast address, and to not expect replies when sending commands to unit zero. However, many devices do not obey this part of the specfication, so this module will allow definition of unit zero as a real node.

Modbus OPC Addressing

The OPC addressing formats for both of this module's driver types (client and server) are compatible with the addressing documented for Ignition's native Modbus client driver, but with additional features (Bit fields, eXtended Registers).

Supported Address Formats

Pattern	Range	Data Type	Description
C*n*	0 ≤ n ≤ 65535	Boolean	Memory Area '0', corresponding to 000001 through 065536.
Cn:m[1]	0 ≤ n ≤ 65535, 1 ≤ m ≤ 63	UInt64 => int8	Memory Area '0', consecutive bits interpreted as a little-endian field.
DI*n*	0 ≤ n ≤ 65535	Boolean	Memory Area '1', corresponding to 100001 through 165536.
DIn:m[1]	0 ≤ n ≤ 65535, 1 ≤ m ≤ 63	UInt64 => int8	Memory Area '1', consecutive bits interpreted as a little-endian field.
IR*n*	0 ≤ n ≤ 65535	Int16 => int2	Memory Area '3', corresponding to 300001 through 365536.
HR*n*	0 ≤ n ≤ 65535	Int16 => int2	Memory Area '4', corresponding to 400001 through 465536.
XR*n*[2]	0 ≤ n ≤ 655350000	Int16 => int2	Memory Area '6', corresponding to 6000000001 through 4655360000.
Memory areas 3, 4, and 6 share multiple patterns. Use *n* as above except as noted, and prefix *x*="I", "H", or "X" for the corresponding memory area:			
*x*R*n.m*[3]	0 ≤ m ≤ 15	Boolean	Single bit or word.
*x*R*n.m:s*[1,3]	0 ≤ m ≤ 15, 1 ≤ s ≤ 15	Int16 => int2	Consecutive bits of word interpreted as an LE field.
*x*RBCD*n*		Int16 => int2	Converted from 4-digit Binary Coded Decimal.
*x*RUS*n*		UInt16 => int4	Interpreted as unsigned.
*x*RF*n*[4]		Real32 => float4	Two consecutive words interpreted as IEEE 754 32-bit floating point.
*x*RI*n*[4]		Int32 => int4	Two consecutive words interpreted as a 32-bit integer.
*x*RUI*n*[4]		UInt32 => int4	Two consecutive words interpreted as an unsigned 32-bit integer.
*x*RIBCD*n*[4] *x*RBCD_32*n*[4]		UInt32 => int4	Two consecutive words converted from 8-digit Binary Coded Decimal. (Either form accepted.)
*x*RMI[4,5]		Int32 => int4	Two consecutive words converted from Schneider "Mod10, size=2" format.
*x*RM[4,5]		Int48 => int8	Three consecutive words converted from Schneider "Mod10, size=3" format.
*x*RML[4,5]		Int64 => int8	Four consecutive words converted from Schneider "Mod10, size=4" format.
*x*RD*n*[4]		Real64 => float8	Four consecutive words interpreted as IEEE 754 64-bit floating point.
*x*RL*n*[4] *x*RI_64*n*[4]		Int64 => int8	Four consecutive words interpreted as a 64-bit integer. (Either form accepted.)
*x*RUL*n*[4] *x*RUI_64*n*[4]		UInt64 => int8	Four consecutive words interpreted as an unsigned 64-bit integer. (Either form accepted.)
*x*HRS*n:m*[4,6]	1 ≤ m ≤ 246 (subject to single read word count limit)	String	One or more consecutive words interpreted as pairs of ASCII characters. 'm' is the number of characters. If m is odd, the last half of the last word is ignored.

Notes:

[1] Not shown in OPC browse.

[2] Files in Modbus are groups of 10,000 16-bit registers. The protocol allows files numbered 1 through 65535, for a storage area having up to 655,350,000 registers.

[3] Bits and bit fields are written by a client driver using function code 20, Masked Write, which only works with single words of memory area '4', Holding Registers. In the client driver, these address formats are read-only in memory areas '3' and '6'.

[4] Multiword data types may **not** span unconfigured addresses nor wrap around the end of the area, or for memory area '6', cross a file boundary. See the "Configuration" section of the drivers for more information.

[5] The registers in "Mod10" format are each expected to hold a value from 0 to 9,999 if unsigned, or ±9,999 if signed. See the Schneider documentation here.

[6] The string address format is shown in OPC browse operations with length "2". Adjust the OPC item to the desired length after drag-n-drop, or construct the OPC path manually.

Advanced Modbus Server

Ignition's native Modbus driver module is a client only. It only connects as a master to one or more slaves--devices and other systems (servers) that can *respond* to Modbus requests. This Advanced Modbus Driver module includes slave functionality. Since the protocol is the same, Ignition gateways running this driver module can accept connections from other Ignition gateways running the native Ignition driver.

This driver constructs storage areas in memory, per configured slave unit and with configured sizes, corresponding to the Modbus Application Protocol's memory areas '0', '1', '3', '4', and '6'. At startup it optionally restores the last saved contents of these storage areas. It saves memory contents to disk upon normal shutdown, and optionally on a configurable time interval.

This driver listens for TCP connections on zero or more specified local addresses, using the expanded protocol header. (The listen address may be the 0.0.0.0 wildcard, which listens on all interfaces.) This driver can set up and open zero or more local (to the Ignition gateway) serial ports, and can make zero or more raw RTU over TCP connections to remote serial ports. All of these connection types may be used simultaneously. (Automation Professionals recommends Moxa NPort products in server mode for raw RTU over TCP connections.)

This module does not support all function codes in the specification. It focuses on the function codes most popular for data access.

Supported Function Codes

Operation	*Decimal Code*	*Hex Code*
Read Coil Bits, Memory Area '0', Prefix 'C'	1	0x01
Read Discrete Input Bits, Memory Area '1', Prefix 'DI'	2	0x02
Read Holding Register Words, Memory Area '4', Prefix 'HR'	3	0x03
Read Input Register Words, Memory Area '3', Prefix 'IR'	4	0x04
Write Single Coil Bit, Memory Area '0', Prefix 'C'	5	0x05
Write Single Holding Register Word, Memory Area '4', Prefix 'HR'	6	0x06
Write Coil Bits, Memory Area '0', Prefix 'C'	15	0x0f
Write Holding Register Words, Memory Area '4', Prefix 'HR'	16	0x10
Read File Record Words, Memory Area '6', Prefix 'XR'	20	0x14
Write File Record Words, Memory Area '6', Prefix 'XR'	21	0x15
Masked Write Single Holding Register Word, Memory Area '4', Prefix 'HR'	22	0x16

For the multiple element reads and writes listed above, the maximum quantities given in the protocol specification are supported.

Note that the OPC driver connection to the emulated slave units bypasses these function codes, so it may write to any data area. (Memory areas '1' and '3' would be useless if Ignition itself couldn't write to them.) This also means the OPC driver connection can *write* booleans and bit fields in *all* memory areas.

Data Consistency

With multiple simultaneous client connections, along with the OPC driver connection to Ignition itself, it is possible for multiple read and/or write requests to target the same bit or words in the same unit at the same time. This driver avoids data corruption by synchronizing access to each memory area in each configured unit. Only one request at a time will be allowed access to that memory area. As a further optimization for the OPC driver connection, any operations submitted together will be sorted by unit and memory area, then the corresponding lock held while those operations execute together.

Modbus TCP

For this type of inbound connection, Ignition will open a standard bound listening TCP socket at the specified addresses (optionally with a TCP port number) and will spawn actual TCP connections upon demand. There is no artificial limit on the number of simultaneous TCP connections. Each actual connection will allow multiple simultaneous requests and will use the header transaction IDs to manage potential out-of-order responses. Note: This could result in a denial of service condition if exposed to nefarious actors.

Modbus RTU (Local and Remote Serial)

Serial ports, both local and remote, will be treated as RS-485 connections and will ignore requests/replies involving slave unit numbers that aren't configured locally. This allows Ignition to co-exist with other slave devices on a multi-drop RS-485 bus. For this to work correctly, Ignition must know the serial port timing properties involved. For local serial ports, the baud rate, parity, stop bits, and flow control are set to the specified values when opening the port. For remote serial ports via raw TCP, the specified values must match the remote port's actual configuration.

Device Settings

Each driver instance (Ignition "Device") functions as one or more virtual slave devices. (How many slaves are emulated and what each contains is described in the next section, Device Units Configuration.) At driver instance creation, a single slave unit, address #1, is created, pre-configured with all possible addresses in memory areas '0', '1', '3', and '4'.

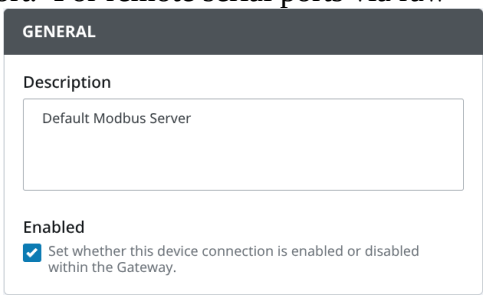


Figure 1: Server Device General Settings

General

These are the settings all drivers have, required by the OPC Server itself.

Communications

These settings control how the driver instance communicates with external clients.

TCP Listening Addresses will use conventional bound sockets accepting multiple simultaneous connections. The following formats are accepted (optional parts in square brackets):

- `hostname[:port]`
- `ip_address[:port]`

In a redundant server pair, the backup server will use the list of addresses in *Backup Listening Addresses* instead. In a redundant pair, only the Active server will accept connections.

RTU Listening Ports identifies the local and/or remote serial ports that this driver instance will take control of and listen to. The following formats are accepted (optional parts in square brackets):

- `hostname:port:baud[parity][stop]`
- `ip_address:port:baud[parity][stop]`
- `COMn:baud[parity][stop][flow]`
- `/dev/ttyXn:baud[parity][stop][flow]`

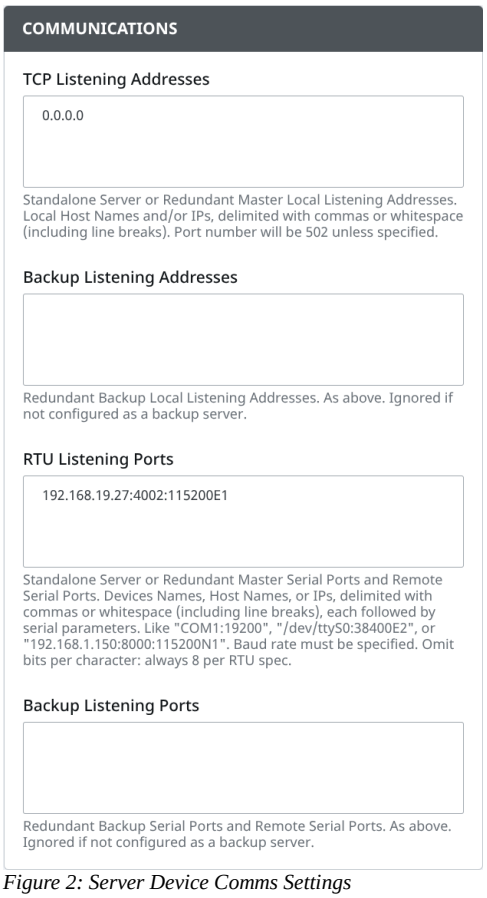


Figure 2: Server Device Comms Settings

In a redundant server pair, the backup server will use the list in *Backup Listening Ports* instead. In a redundant pair, only the Active server will open the given ports.

Drop Idle TCP Delay sets an idle timeout (in milliseconds) for traffic on any Modbus TCP connection to the server. The default is ten seconds, matching the behavior of many devices in practice. Values below five seconds are clipped to five seconds.

In the above RTU connection targets, “baud” is anything over 300 that is physically supported. “parity” is a single letter indicating Even, Mark, None, Odd, or Space—Even parity is the default if omitted. “stop” is “1”, “15”, or “2”, where “15” represents 1½ stop bits—“1” is the default. For real physical ports, “flow” is “H”, “C”, “R”, “DS”, “DT”, or “N”. “H” and “R” both represent RTS flow control. “C” is RTS+CTS, “DS” is DSR-only, “DT” is DSR+DTR, and “N” is None. None is the default. Linux and MacOS only support “N” or “C”. Windows supports all of the above. Software flow control is forbidden by the Modbus specification. A colon or hyphen may be used as a delimiter between baud, parity, and stop bit if ambiguous.

Each RTU connection will only process one request at a time, and the master on the channel is expected to follow the specification for RTU timing with RS-485. Other slaves are allowed on the physical connection, as master requests and slave replies with other slave unit numbers will be totally ignored.

Behavior

The *Drop Idle TCP Delay* setting is the number of milliseconds to hold an incoming TCP connection open while it is idle (issues no requests). It is common for industrial devices for force-close TCP connections that sit idle for ten seconds. Set this to MAX_INTEGER or other very large value to effectively disable this behavior.

If *Use Persistence* is checked, this driver will save the memory contents of each configured slave in binary files in the device’s home directory. This folder is located in your Ignition install folder, under `.../data/drivers/<device Name>`. Files are saved on device shutdown, and restored upon device startup. By default, they are also saved every thirty seconds. If *Persist Interval* is set to zero, save while running is disabled.

No attempt is made to perform any other management of the binary snapshot files. Any missing snapshots will simply yield zeros in the corresponding slave memory upon startup. Files orphaned by unit deletion or renumbering will remain until manually deleted.

The *Extra Serial RTU Framing* and *Extra TCP RTU Framing* settings are discussed in the Appendix for application notes.

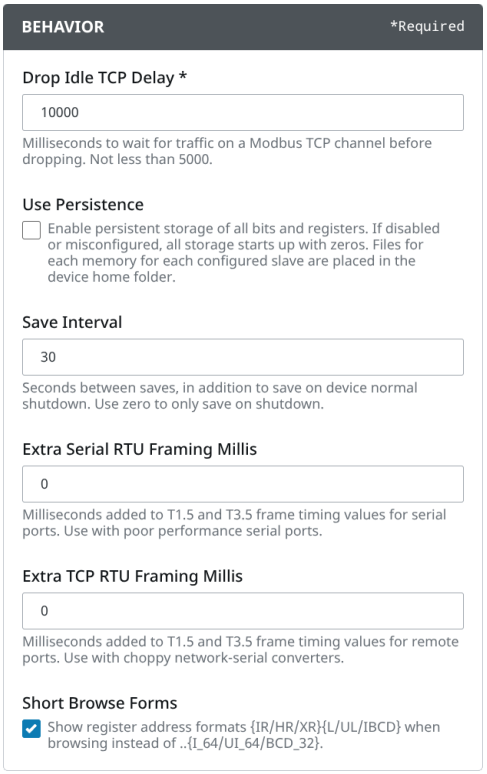


Figure 3: Server Device Behavior Settings

The *Short Browse Forms* setting alters the presentation of nodes and node paths in the OPC Browser. When checked (the default), this module avoids the underscores and shows this driver's shorter forms. Both short and long forms are always ***accepted*** in OPC Item Paths.

Device Units Configuration

The number and addresses and properties of the slave units to be emulated are not editable in the device’s Settings page. A separate page is provided, under the Configuration menu item of each device, to alter the list of slave units from your device’s emulation.

In this version of the driver, editing of the unit list and the per-unit settings is only supported via CSV file.

Import/Export

The import/export format is a relaxed syntax CSV, where string elements that do not need quotes for correctness may omit the quotes. Exports include a header line, and it is required on import to indicate the columns present and their order. Missing columns, other than UnitId, are not an error on import, and the default will be used for all units.

Use the *Export Units CSV* button at top of the Configuration page to obtain the currently configured Units CSV, as originally provided for import. If disabled, no configuration has been imported.

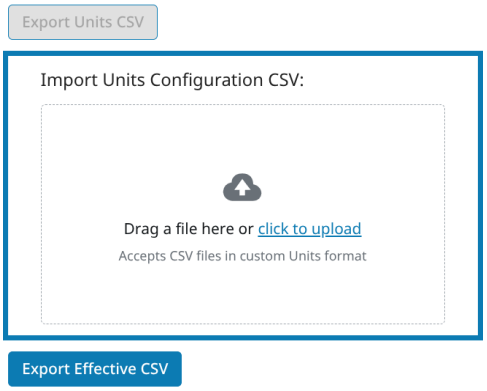


Figure 4: Device Units Configuration

Use the *Import Units Configuration CSV* upload area or link to supply a configuration file in the format described below. The format will be syntax checked, and applied as-is if valid. The imported file ***replaces*** any prior configuration, and the device will restart ***immediately***.

Use the *Export Effective CSV* button at the bottom of the Configuration page to obtain the configuration CSV with all columns and defaults present. When no configuration has been imported, this export will show the settings for the implicit Unit #1.

Be aware that some field values only require quotes when multiple entries are present in that field (because they then contain a comma). You may need to add quotes for such fields when you edit or generate the CSV content directly.

Server Unit Detail Configuration CSV Columns

The first line of a configuration CSV column must contain the column names for the following lines. Columns other than UnitId can be omitted. Any columns with names other than those in the table below will be discarded without error. (Enables config file sharing between client and server device types.)

Column Name	*Description*	*Default*	*Implicit Node #1*
UnitId	Must be an integer in the range of zero to 255.		1
SwapChars	When true, xRS formats will operate in little-endian mode.	false	false
SwapWords	When true, xRI and xRL formats (and variants) will operate in little-endian mode, by 16-bit word. (16-bit words are always big endian in Modbus.)	false	false
SwapFloat	When true, xRF and xRD formats (and variants) will operate in little-endian mode, by 16-bit word. (16-bit words are always big endian in Modbus.)	false	false
CoilsRange	List of single registers or register ranges to expose to external clients and to allow in OPC Item Paths. All registers must be within the 0-65535 range of values that are valid in the Modbus protocol. When multiple ranges are present, use commas to separate them, and use double quotes in the CSV file. All multi-word operations, externally and via OPC, will not be allowed to cross any range boundary.	{empty}	0-65535
DiscretesRange			0-65535
InputRegRange			0-65535
HoldingRegRange			0-65535
ExtendedRegCount	Quantity of File Registers to support.	0	98304

Automated Device Creation

Instances of this device type can be constructed with Ignition's native `system.device.addDevice()` scripting function. Use "ModbusServer" as the device type ID. Use the following keys in the device properties argument:

Key	Data Type	Content
comms.masterTCP	String	Modbus TCP local listening IP address:port combinations, one per line.
comms.masterRTU	String	Modbus RTU local serial and/or remote TCP connections, one per line.
comms.backupTCP	String	Same functionality as "mastertcp", but applied to the backup server of a redundant pair.
comms.backupRTU	String	Same functionality as "masterrtu", but applied to the backup server of a redundant pair.
behavior.dropIdleTCP	Boolean	Milliseconds to allow idle Modbus TCP clients to exist without traffic.
behavior.persist	Boolean	When true, save server content to binary files to establish initial values.
behavior.persistInterval	Integer	When greater than zero, auto-save server content every "x" seconds.
behavior.tFramePlusSer	Integer	Extra Serial RTU Framing Milliseconds
behavior.tFramePlusTcp	Integer	Extra RTU over TCP Framing Milliseconds
behavior.shortBrowse	Boolean	When true, simplify browse of certain 64-bit types

All keys are ***case-sensitive***. One of "comms.masterTCP" or "ccomms.masterRTU" is required. All others have sane defaults.

The units configuration file must be delivered via API at this time, to the auxiliary resource file "units.csv" in the device configuration resource.

Advanced Modbus Client

This single driver type handles all Modbus Client connections, with multiple slave units configurable. It supports all of the function codes and addressing features of the Server Driver, enabling symmetric communications between Ignition gateways with this module.

During bursts of OPC requests, this client will round-robin through the slave units (if multiple slaves are present). This maximizes utilization if any slaves have processing performance limits, and prevents starvation of any given unit. Each unit can also be throttled by configuring a minimum inter-request interval—it will be skipped in the rotation when the throttle applies.

As a further optimization, if a request times out, all of that slave unit's queued requests will be canceled, too. This gives that slave unit's peers an opportunity to execute without repeated timeouts from a failed/disconnected slave.

Supported Function Codes

Operation	*Decimal Code*	*Hex Code*	*Config Notes*
Read Coil Bits, Memory Area '0', Prefix 'C'	1	0x01	Qty Limit
Read Discrete Input Bits, Memory Area '1', Prefix 'DI'	2	0x02	Qty Limit
Read Holding Register Words, Memory Area '4', Prefix 'HR'	3	0x03	Qty Limit
Read Input Register Words, Memory Area '3', Prefix 'IR'	4	0x04	Qty Limit
Write Single Coil Bit, Memory Area '0', Prefix 'C'	5	0x05	Used only when code 15 limit is 0
Write Single Holding Register Word, Memory Area '4', Prefix 'HR'	6	0x06	Used only when code 16 limit is 0
Write Coil Bits, Memory Area '0', Prefix 'C'	15	0x0f	Qty Limit
Write Holding Register Words, Memory Area '4', Prefix 'HR'	16	0x10	Qty Limit
Read File Record Words, Memory Area '6', Prefix 'XR'	20	0x14	
Write File Record Words, Memory Area '6', Prefix 'XR'	21	0x15	
Masked Write Single Holding Register Word, Memory Area '4', Prefix 'HR'	22	0x16	If enabled, Booleans and Bit Fields are Writeable

If the quantity limit for reading a given memory area is zero, or no addresses are configured "present", the entire memory area will be omitted from the browse for the slave unit, and the entire memory area will be unavailable. If the quantity limit for either register memory area is less than four, 64-bit data types will not be available. If limited to one, 32-bit types will not be available. Similarly, if writes to holding registers are limited to less than four, 64-bit types will be read-only. If less than two, 32-bit types will be read-only. As a special case, when the quantity limit for function codes 15 or 16 are set to zero, the corresponding single-element function code will be used for all writes.

In addition to the function codes supported as OPC items, client connections can use any arbitrary function code with any suitable byte array payload. See Scripting Arbitrary Modbus Requests, below, for details.

Device Settings

Each driver instance makes a single connection. The target may have multiple slave units, just like the native Ignition driver. Unlike the native driver, this driver will reject requests for units that are not configured. How many slave units are configured and what each contains is described in the section below. At driver instance creation, a single slave unit, address #1, is created, pre-configured with all possible addresses in memory areas '0', '1', '3', and '4'.

General

These are the settings all drivers have, required by the OPC Server itself.

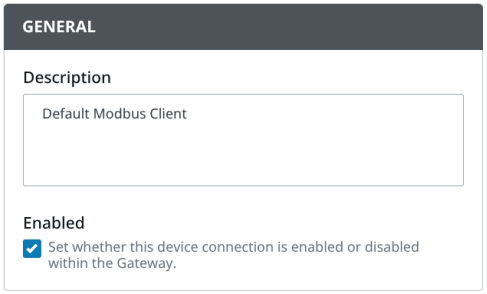


Figure 5: Client Device General Settings

Communications

This section selects the type of connection and the target of the connection. The type may be Modbus TCP, Modbus RTU Serial, or Modbus RTU via TCP. The following formats are accepted for Modbus TCP:

- `hostname[:port]`
- `ip_address[:port]`

The default port for Modbus TCP is 502.

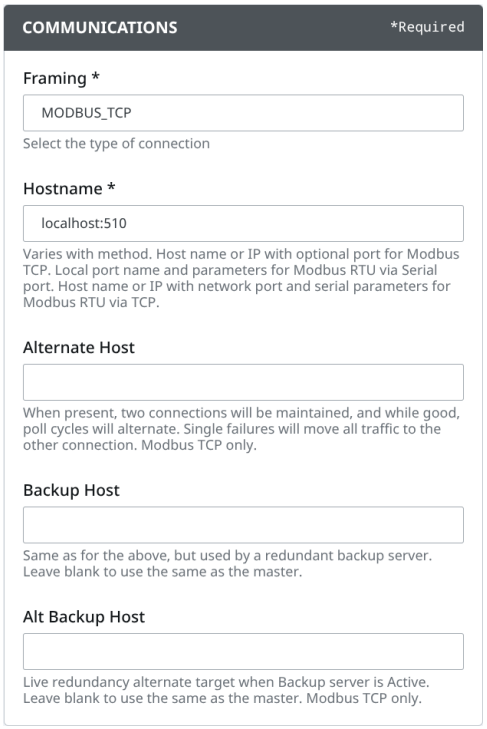
The following formats are accepted for Modbus RTU Serial:

- `COMn:baud[parity][stop][flow]`
- `/dev/ttyXn:baud[parity][stop][flow]`

The following formats are accepted for Modbus RTU via TCP:

- `hostname:port:baud[parity][stop]`
- `ip_address:port:baud[parity][stop]`

For a description of the parameters in RTU connections, see the description of Behavior in the Server Driver section above.



Live redundancy is supported for Modbus TCP by specifying and additional target/port combination to be opened simultaneously by the driver. The driver will alternate connections for small batches of requests as long as both are responding. This is not strictly the same as load-sharing, but behaves similarly. Requests already queued to a connection when breakage is detected will receive an error response—there is no automatic requeuing.

A target/port combination may be specified separately for the Backup Server in a redundant pair. If left blank, the redundant pair will connect to the same target as the Master Server. When using live redundancy with server redundancy, the backup server may also have a different target/port combination for live redundancy.

Modbus TCP

For this type of connection, Ignition will open a standard connected TCP socket to the specified addresses (optionally with a TCP port number) and begin issuing requests as demanded by the OPC server. If the Concurrent Requests setting is greater than one, that many requests will be issued without stalling for a matching reply, and replies may indicate out-of-order processing with the protocol's transaction IDs.

As a special case for non-compliant devices, if the Concurrent Requests setting is equal to one, any mismatch between the transaction ID sent and that in the reply will be ignored.

Modbus RTU Serial

A local serial port will be setup with the given timing and framing properties and opened on startup, and then treated as an RS-485 connection. Reply framing and timing per the specification will be enforced. Non-compliant replies will be discarded (generally resulting in a timeout).

Modbus RTU via TCP

A raw TCP connection to a remote serial port will be opened on startup, and then treated as an RS-485 connection. The timing and framing properties specified with the target address/port must match the remote port's actual configuration. Reply framing and timing per the specification will be enforced. Non-compliant replies will be discarded (generally resulting in a timeout).

Behavior

Concurrent Requests and *Ignore TxID* may be used with Modbus TCP. If the latter is true, *Concurrent Requests* is ignored and treated as "1".

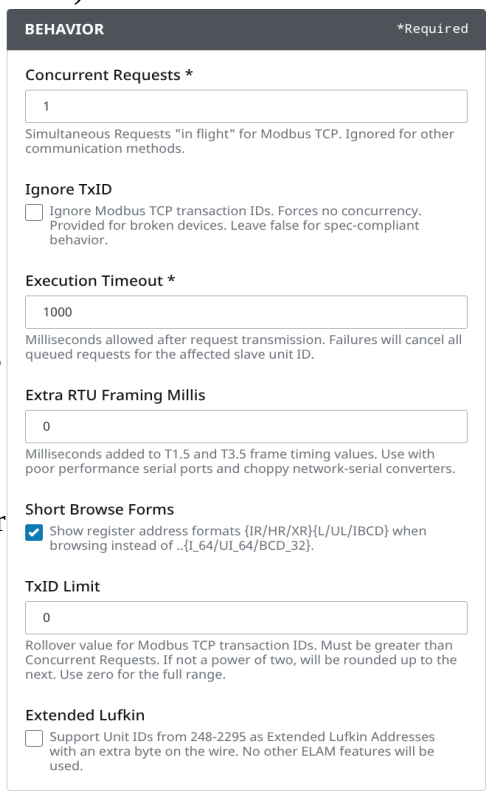
The *Execution Timeout* for a connection, which applies to all units, is configurable.

See the RTU Frame Timing Adjustment discussion in the appendix for details of the *Extra RTU Framing* setting.

If the driver instance needs to offer browse addresses that will later be used with Inductive Automation's native Modbus driver, you should turn off the *Short Browse Forms* setting.

The *Transaction ID Limit* setting offers a work-around for broken Modbus TCP devices that do not allow use of the full 16-bit integer range for transaction IDs.

The *Extended Lufkin* setting is for compatibility with RTU slave units that follow Lufkin's "ELAM" extension to Modbus. The Ignition driver will not use multi-function streaming requests, nor use request sizes beyond the normal standard, but will encode/decode addresses with the extra byte in the header.



Device Units Configuration

The number and addresses and properties of the slave units to be emulated are not editable in the device's Settings page. A separate page is provided, under the Configuration menu item of each device, to alter the list of slave units from your device's emulation.

In this version of the driver, editing of the unit list and the per-unit settings is only supported via CSV file. This is identical to the Import/Export for the Server Device described above.

Import/Export

The import/export format is a relaxed syntax CSV, where string elements that do not need quotes for correctness may omit the quotes. Exports include a header line, and it is required on import to indicate the columns present and their order. Missing columns, other than UnitId, are not an error on import, and the default will be used for all units.

Use the *Export Units CSV* button at top of the Configuration page to obtain the currently configured Units CSV, as originally provided for import. If disabled, no configuration has been imported.

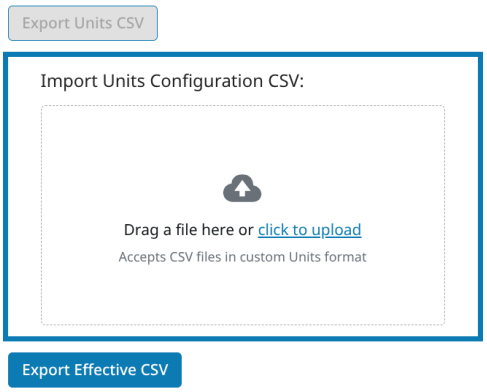


Figure 6: Device Units Configuration

Use the *Import Units Configuration CSV* upload area or link to supply a configuration file in the format described below. The format will be syntax checked, and applied as-is if valid. The imported file ***replaces*** any prior configuration, and the device will restart ***immediately***.

Use the *Export Effective CSV* button at the bottom of the Configuration page to obtain the configuration CSV with all columns and defaults present. When no configuration has been imported, this export will show the settings for the implicit Unit #1.

Be aware that some field values only require quotes when multiple entries are present in that field (because they then contain a comma). You may need to add quotes for such fields when you edit or generate the CSV content directly.

Client Unit Detail Configuration CSV Columns

The first line of a configuration CSV column must contain the column names for the following lines. Columns other than UnitId can be omitted. Any columns with names other than those in the table below will be discarded without error. (Enables config file sharing between client and server device types.)

Column Name	*Description*	*Default*	*Implicit Node #1*
UnitId	Must be an integer in the range of zero to 255.		1
SwapChars	When true, xRS formats will operate in little-endian mode.	false	false
SwapWords	When true, xRI and xRL formats (and variants) will operate in little-endian mode, by 16-bit word. (16-bit words are always big endian in Modbus.)	false	false
SwapFloat	When true, xRF and xRD formats (and variants) will operate in little-endian mode, by 16-bit word. (16-bit words are always big endian in Modbus.)	false	false
CoilsRange	List of single registers or register ranges to expose to external clients and to allow in OPC Item Paths. All registers must be within the 0-65535 range of values that are valid in the Modbus protocol. When multiple ranges are present, use commas to separate them, and use double quotes in the CSV file. All multi-word operations, externally and via OPC, will not be allowed to cross any range boundary.	{empty}	0-65535
DiscretesRange			0-65535
InputRegRange			0-65535
HoldingRegRange			0-65535
ExtendedRegCount	Quantity of File Registers to support.	0	98304
SpanGaps	When true, allow reads of otherwise unused addresses. See Read Request Optimization below.	true	true
MaxReadCoil	The largest number of coils allowed to be read using Function 0x01.	2000	2000
MaxReadDiscrete	The largest number of discrete inputs allowed to be read using Function 0x02.	2000	2000
MaxReadHolding	The largest number of holding registers allowed to be read using Function 0x03.	125	125

Column Name	*Description*	*Default*	*Implicit Node #1*
MaxReadInputReg	The largest number of inputs registers allowed to be read using Function 0x04.	125	125
UseMultipleGroups	When true, the optimizer may request reads or writes to fragments of multiple files in a single request. (Functions 0x14 and 0x15.)	true	true
MaxWriteCoil	Maximum number of coils to be written with Function 0x0f. If zero, all coils writes will use Function 0x05. See Write Request Options below.	1968	1968
MaxWriteHolding	Maximum number of holding registers to be written with Function 0x10. If zero, all coils writes will use Function 0x06. See Write Request Options below.	123	123
UseMaskedWrite	Allow the use of Function 0x16 to write single bits in holding registers. See Write Request Options below.	true	true
InterRequestDelay	Prevent two requests to this same unit within the given number of milliseconds.	0	0

Read Request Optimization

The *Span Gaps* option is available to prevent the the driver from combining requests that read nearby addresses if there would be intervening addresses not used. In this driver, this is not needed if troublesome addresses are omitted from the configured range.

Read requests for memory areas ‘0’, ‘1’, ‘3’, and ‘4’ may be restricted to smaller quantities than the specification would normally allow. If tightly restricted, this may also impact some datatype support, which would be reflected in the OPC Browse for the slave unit. See the Supported Function Codes section for details.

Write Request Options

Write requests for memory areas ‘0’ and ‘4’ may be restricted to smaller quantities than the specification would normally allow. If tightly restricted, this may cause some data types to be read-only.

The Masked Writes option may be disabled for slave units that do not support it. In this case, the bits and bit fields of Holding Registers (memory area ‘4’) will be read-only.

Automated Device Creation

Instances of this device type can be constructed with Ignition's native `system.device.addDevice()` scripting function. Use "ModbusClient" as the device type ID. Use the following keys in the device properties argument:

Key	Data Type	Content
comms.framing	Enum	One of "MODBUS_TCP", "MODBUS_RTU_SERIAL", or "MODBUS_RTU_OVER_TCP"
comms.hostName	String	Target in the format required by the chosen framing.
comms.altHost	String	Same functionality as "hostname", used to open a live redundant channel.
comms.backupHost	String	Same functionality as "hostname", but applied on the backup gateway of a redundant pair.
comms.altBackupHost	String	Same functionality as "althost", but applied on the backup gateway of a redundant pair.
behavior.maxOutstanding	Integer	Concurrency for Modbus TCP
behavior.ignoreTxId	Boolean	Fix for certain broken Modbus TCP gateways.
behavior.execTimeout	Integer	Milliseconds to wait for responses, per request.
behavior.tFramePlus	Integer	Extra RTU Framing Milliseconds
behavior.shortBrowse	Boolean	When true, simplify browse of certain 64-bit types
behavior.txIdLimit	Integer	Transaction ID rollover limit, to accommodate broken devices
behavior.elamSupport	Boolean	Enable Extended Lufkin Address Mode on RTU connections.

All keys are ***case-sensitive***. Just "comms.hostName" is required. All others have sane defaults.

The units configuration file must be delivered via API at this time, to the auxiliary resource file "units.csv" in the device configuration resource.

Scripting Arbitrary Modbus Requests

Client connections may process arbitrary requests using the `rawModbus` function in Gateway Scope.

This function is normally asynchronous, returning a standard java CompletableFuture. The caller would use the `.get()` method, or one of the other standard asynchronous forms, to obtain the response.

The response will be an Ignition QualifiedValue, with one of the following formats:

- Success. The quality will be good and the value will be a byte array containing the actual response.
- Failure, reported by the target device. The quality will be bad, of a suitable type if possible. The value will be an integer containing the error code supplied by the target device.
- Failure, reported by Ignition. The quality will be bad, of a suitable type. The value will be null/None.

Figure 7: system.opc.rawModbus(name, unit, func, payload, rspLen)

Submits the Modbus request constructed from the given unit, function code, and payload to the named device connection. The response must have the specified length (after the echoed function code).

Returns a java CompletableFuture, that will yield an Ignition QualifiedValue when the response is received or an error occurs.

Argument	*Data Type*	*Description*
name	String	OPC Server Device Name
unit	int	Slave Unit Number to target. Does **not** have to be a configured node. 0-255
func	int	Modbus Function Code. 1-127
payload	byte[]	Request content. Supply a zero length array if not needed. Up to 252 bytes.
rspLen	int	Number of reply payload bytes. 0-252

Keyword-style invocation is not allowed. All arguments are required.

This function can also throw an immediate Exception, particularly when the named device doesn't exist or isn't enabled.

The following example shows how to re-implement a read of HR4 through HR10 with function code 3.

Figure 8: Example use of system.opc.rawModbus()

```
from java.io import ByteArrayInputStream, ByteArrayOutputStream, DataInputStream, DataOutputStream
from com.inductiveautomation.ignition.common.model.values import QualifiedValue

# Raw implementation of read multiple registers, function 3.

def printHR4to9():
     # Start by constructing the payload of the Modbus PDU.  It is
     # a UINT starting address and a UINT number of registers (<=125).
     baos = ByteArrayOutputStream()
     dos = DataOutputStream(baos)
     dos.writeShort(4)
     dos.writeShort(6)
     dos.flush()

     future = system.opc.rawModbus('someDevice', 1, 3, baos.toByteArray(), 13)
     # The following blocks until the response comes back or the request times out.
     QV = future.get()

     if QV.quality.good:
          bais = ByteArrayInputStream(QV.value)
          dis = DataInputStream(bais)
          print "Bytes to follow=%d" % dis.readByte()
          print "HR4: %6d" % dis.readShort()
          print "HR5: %6d" % dis.readShort()
          print "HR6: %6d" % dis.readShort()
          print "HR7: %6d" % dis.readShort()
          print "HR8: %6d" % dis.readShort()
          print "HR9: %6d" % dis.readShort()
     else:
          print repr(QV)
```

Moxa NPort Application Notes

The Moxa family of remote serial port devices is popular, and is used in Automation Professionals' test lab. In general, remote serial ports using TCP connections must balance character buffering against latency. The default settings for a port in Moxa's "TCP Server" mode are not suitable for Modbus RTU timing. Specifically, if a Modbus request or reply received by the physical serial port is split into two separate TCP packets, the timing criteria for Modbus RTU are unlikely to be satisfied. To make it accumulate a complete request or reply, and transmit it as a single packet, set the Moxa Port's "Force Transmit" setting to the millisecond value from the following table:

Baud Rate	***Force Transmit Milliseconds***
300	55
600	28
1,200	14
1,800	10
2,400	7
4,800	4
7,200	3
9,600	2
≥ 19,200	1

The calculation is $\frac{1000\,ms}{baud} * 11 * 1.5$, rounded **up**, from the T1.5 quiet factor in the Modbus Specification.

See also the following section on Serial Ports and the possible use of the "Extra RTU Framing Milliseconds" device setting(s).

Linux Serial Port Application Notes

The Linux Operating System's standard for interfacing with system serial ports has unfortunate legacy behaviors that prevent those ports from working well "out of the box". The jSerialComm library has worked around some of these limitations, but external adjustments are needed, too.

Low Latency Mode

First, the serial devices must be placed in low latency mode via the `setserial` utility. The most common commands would look something like:

```
setserial /dev/ttyS0 low_latency
```

or

```
setserial /dev/ttyUSB0 low_latency
```

For USB devices, this command will be needed every time the port is unplugged/reconnected. A rule to make `udev` do this for you would look something like this:

```
KERNEL=="ttyUSB0", RUN+="/bin/setserial %E{DEVNAME} low_latency"
```

(Save that as "`/etc/udev/rules.d/97-usb-serial-low-latency.rules`" or similar.)

Serial Device Access Privileges

Next, the ignition user account (or whatever user runs the Ignition gateway service) must be added to the group controlling access to these devices. On most Linux distributions, this is the `dialout` group. This will work in most cases (executed as root):

```
adduser ignition dialout
```

Be sure to restart Ignition after changing group membership—it doesn't take immediate effect.

RTU Frame Timing Adjustment

Finally, within Ignition, if you are having trouble with timeouts waiting for responses, temporarily set the ModbusRTUComms logger to DEBUG and look for "Discarding packet due to gap". Adjust the Client Mode device setting "Extra RTU Framing Milliseconds" to accommodate the actual latency for your situation. In Server mode, there are separate settings for RTU over TCP versus local Serial Ports.