Ignition EtherNet/IP Module User Manual

EtherNet/IP Communications Suite

Version 2.2.4-v81

October 9, 2026

This EtherNet/IP™ Communications Module for Ignition™ adds multiple OPC driver types to Ignition's native OPC server, with support for Generic EtherNet/IP Client connections, "Class 1" I/O device emulation for use with external I/O scanners, "Class 1" I/O scanner operation for use with external I/O adapter hardware, and a variety of related PLC messaging support.



Table of Contents

1 Overview....5
1.1 New Features of v2.x....5
1.2 Generic Client Driver Features....6
1.3 Generic Host Driver Features (Adapter)....7
1.4 Enhanced Host Driver Features (Scanner)....8
1.5 PLC Feature Compatibility Matrix....10
1.6 Firewall and Networking Requirements....11
1.6.1 Ports....11
1.6.2 Routes....11
2 Client Driver....13
2.1 Settings....13
2.2 Configuration....13
2.3 OPC Interface....14
3 Client Driver Application Notes....14
3.1 Allen-Bradley ControlLogix & CompactLogix....14
3.1.1 Predefined Data Types....15
3.1.2 I/O Module Data Types....15
3.1.3 Add-on Instructions....16
3.1.4 Structure Alignment Rules....16
3.2 Allen-Bradley MicroLogix 800 family....16
3.3 Omron NJ/NX family....17
3.3.1 Structure Alignment Rules....17
3.3.2 Structures with User-defined Member Offsets....17
3.3.3 Omron Strings....18
3.3.4 Omron Booleans....18
3.4 CIP Attribute Access....18
3.4.1 Simple Attributes....19
3.4.2 Complex Attributes....19
3.4.3 OPC Browsing of Attributes....19
3.4.4 Polling I/O Assemblies....19
3.5 Keyence KV family....20
3.5.1 OPC Browsing....21
3.5.2 Communication Buffer Optimization....21
3.5.3 Request Optimization....21
4 Host Driver....22
4.1 Settings....22
4.1.1 Local Listening IP Addresses....22
4.1.2 Backup Listening IP Addresses....22
4.2 Configuration....23
4.2.1 Default Configuration....23
4.2.2 Configuration XML Format....23
4.2.3 Runtime Configuration....24
4.2.4 Data Type Emulation....24
4.2.5 Tag Emulation....24
4.2.6 CIP Assembly Emulation....25
4.2.7 I/O Module Scanning....26
5 Target Driver....26
5.1 Settings....26
5.2 Configuration....26
6 Host and Target Driver Application Notes....28
6.1 Using Passive Target I/O....28
6.1.1 Solo I/O Module Emulation with Logix Processors....28

6.1.2 Multiple I/O Module Emulation with Logix Processors....30
6.1.3 Redundancy Considerations....32
6.2 Using Passive Messaging....32
6.3 Using Producer Tags....32
6.3.1 Producing to Logix Structure Types....33
6.3.2 Producing Data State Change Events....33
6.4 Using Consumer Tags....33
6.4.1 Consuming Logix Structure Types....33
6.4.2 Consuming Data State Change Events....34
6.5 Using Active I/O Scanning....34
6.5.1 Target Module Electronic Keying....34
6.5.2 Target Module Connection Detail....35
6.5.3 Target Module Data....36
6.5.4 Monitoring and Runtime Adjustment....36
6.5.5 Redundancy Behavior....37
6.6 Using the I/O Momentary Button....37
7 Host and Target Drivers' Functional Description....38
7.1 Object Classes, Instances, and Attributes....38
7.2 EtherNet/IP Encapsulation in TCP/IP....39
7.3 EtherNet/IP Encapsulation in UDP/IP....39
8 Element Path Segments....41
8.1 Path Segment Strings and Encodings....41
8.1.1 Element Path String formats....41
8.1.2 Segment Type Tokens Index....42
8.1.3 Segment Type Code Summary....43
8.2 Route Path Segments....44
8.2.1 Port Segments....44
8.2.2 Electronic Key Segments....44
8.2.3 Network Parameter Segments....44
8.3 Application Path Segments....45
8.3.1 Numeric Logical and Extended Logical Segments....45
8.3.2 Indirect Extended Logical Segments....45
8.3.3 Symbol and Data Selection Segments....46
8.3.4 Special Symbol Segments....46
8.3.5 Keyence Application Paths....47
8.4 Data Definition Segments....47
8.4.1 Elementary Data Segments....47
8.4.2 Predefined Structure Segments....48
8.4.3 Array Prefix Segments....49
8.4.4 Structure Definition Segments....50
8.4.5 Indirect Reference Segments....51
8.5 Alternate Syntax....51
8.6 Testing Element Paths....51
9 Interpreting EDS Files....53
9.1 Identity....53
9.2 Connection Options....53
9.3 Connection Data....54
9.4 Assembly Structures....55
10 Scripting Features and Functions....56
10.1 Custom Jython Code Modules....56
10.2 Jython Data Events....57
10.3 CIP Messaging Access....58
11 Troubleshooting....60
11.1 OPC Tag Subscriptions....60

11.1.1 Stale Data from a Host or Target Driver....60
11.1.2 Excessive Subscriptions....60
11.2 Scanner Connection Errors....60
12 Allen-Bradley Logix Firmware Variations....63
12.1 Elementary Data Types....63
12.2 Predefined Structured Types....63

1 Overview

The EtherNet/IP Communications Suite Module enables Inductive Automation's Ignition platform to communicate with select Allen-Bradley™ and Omron™ devices, including processors and input/output modules, with tag-based polling and the robust UDP-based networked I/O protocol subset.

The implementation conforms to ODVA's EtherNet/IP Specification, versions 1.20 through 1.29, along with ODVA's Common Industrial Protocol ("CIP") Specification, versions 3.19 through 3.31.

The module is available with various combinations of features, allowing the cost to be tailored to the end-user's requirements. (All features enabled in Trial mode when unlicensed.) The variants are:

- Generic EtherNet/IP Client Driver, for polling Allen-Bradley's ControlLogix™, CompactLogix™, and MicroLogix™ 800 processor families, Omron's NJ/NX processor family, and parameterized EtherNet/IP devices with Electronic Data Sheet files (not brand-specific). This driver is a classic request/response driver using EtherNet/IP "Class 3" TCP/IP **explicit messaging** connections. This driver browses the target device for tag symbols, structure types, and standard class instances.
- Generic EtherNet/IP Host Driver permitting external PLCs to treat Ignition as one or more networked I/O **adapters**, implementing the "target" end of EtherNet/IP "Class 1" UDP/IP **implicit messaging** connections. One or more local IP addresses may be used as listeners, supporting multiple subnets on the Ignition Gateway using . These two drivers transfer I/O assembly buffers to/from structures and arrays and singleton tags modeled after a Logix processor's hierarchical tags. These two drivers will respond to external devices' explicit messaging connections, to read or write the defined virtual tags and a variety of class instances. These two drivers will also implicitly **produce** virtual tags upon external request.
- The Target Driver permits multiple virtual I/O modules to appear in Ignition's virtual chassis using a shared configuration. It is otherwise a subset of the Host Driver and is configured with similar XML. It does not support listening on the network, so must always be paired with at least one Host Driver.
- Enhanced EtherNet/IP Host Driver with I/O **scanner** and message originator features. This is a superset of the Host Driver above, permitting Ignition to directly connect to a wide variety of networked I/O adapters. This enhanced driver can also **consume** virtual tags from external devices.

1.1 New Features of v2.x

- The Generic Client Driver is entirely new.
- The v1.x driver's "Base Features" have been renamed to the Generic Host Driver (Adapter) and companion Target Driver. Like the Client Driver, these drivers support many more CIP data types, along with more options for element alignment rules within CIP structured types. The editors for tags and types have been altered to accommodate these new type features.
- The v1.x driver's "Premium Features" have been renamed to the the Enhanced Host Driver (Scanner). This driver no longer uses an encrypted feature code to control scanner functionality. It is now all-inclusive, controlled by an option in the module license. There are no longer any subnet quantity limits or limits on which Host Drivers can be scanners.
- In v2.1+, the Client Driver supports Class/Instance/Attribute polling of targets, using data types for common classes or from supplemental configuration. Other attributes will be treated as byte arrays.

1.2 Generic Client Driver Features

The generic Client Driver is a request/response messaging driver. It makes a TCP/IP connection as an originator to a target device using the EtherNet/IP protocol, follows an optional CIP route path from the first hop to the target device, probes that target device for available data, and exposes the result in Ignition's OPC Browser (or Quick Client). The possible browsed items include:

- Allen-Bradley ControlLogix and CompactLogix processor families' tags and data types, both controller-scope tags and program-scope tags, but excluding tags with no external access.
- MicroLogix 800 family's global tags, excluding structured types, and excluding arrays where any subscript range is not zero-based. (Limits imposed by the ML800 itself.)
- Omron NJ/NX processor family's global tags and data types, except those set to "Do Not Publish".

This driver is intended to be drop-in compatible with OPC tags currently configured to use Ignition's native Logix v21+ driver, or Ignition's native NJ/NX driver. Where "drop-in" means deleting the original device, then creating an instance of this driver with the same name and communication settings. (In some cases, a gateway restart may be required.) This driver offers more features than the IA driver, which if used, preclude driver substitution in the other direction. Notable items:

- Optimized access to the data structures of Logix Add-On-Instructions, so long as ***none*** of the AOI's members or nested members have external access set to "None". Since that is the default for local tags within AOIs, PLC edits are likely to be required to take advantage of this feature.
- Support for the expanded list of basic data types in recent Logix firmware. These are all data types present in the CIP specification, and this module uses the CIP names for them.
- No need to manually configure a list of tags for an Omron NJ/NX, as is required by Ignition's native driver.
- Optimized access to Omron NJ/NX data structures, except where unions or user-specified custom member offsets are used.
- Optimized access to Omron NJ/NX arrays, including where subscripts are not zero-based.
- Support for the expanded list of basic datatypes in the Omron NJ/NX family, including its "Vendor Specific" BCD, date, time, and date/time data types.
- No need to configure Modbus addresses within MicroLogix 800 family processors, when using compatible tag types.
- Support for one-dimensional numeric arrays in Ignition OPC tags, with lenient write length handling (silent truncation or zero-filling).
- Support for string transforms of array data, allowing numeric arrays (8-bit or 16-bit) to be interpreted as null-terminated strings (with null-filling when written).
- Support for datetime transforms to and from select numeric types.
- Produces Ignition UDT JSON corresponding to the target device's user-defined types, in folders by device name, for convenient import into tag providers. Each has a UDT parameter to drive the OPC Item path of an instance, with proper chaining to nested types. A compact form that leverages array tag types is also offered.
- Produces Ignition Tag JSON corresponding to the target device's entire set of exposed tags, using the UDT definitions for efficiency. This JSON should be pruned to the actually needed items before use. A compact form that leverages array tag types is also offered.

1.3 Generic Host Driver Features (Adapter)

Th ***generic*** Host Driver is a passive ***target***, listening for connections and messages that originate in an external device, typically a Logix™ PLC, Omron NJ/NX, or other PLC with EtherNet/IP Scanner support. (Older Omron PLCs can be EtherNet/IP scanners via option modules.) Ignition appears to be an I/O chassis, with instances of the driver showing up as one or more I/O modules. These virtual I/O modules can also mimic a Logix PLC by responding to symbolic tag data read/write message requests from other systems, including HMIs.

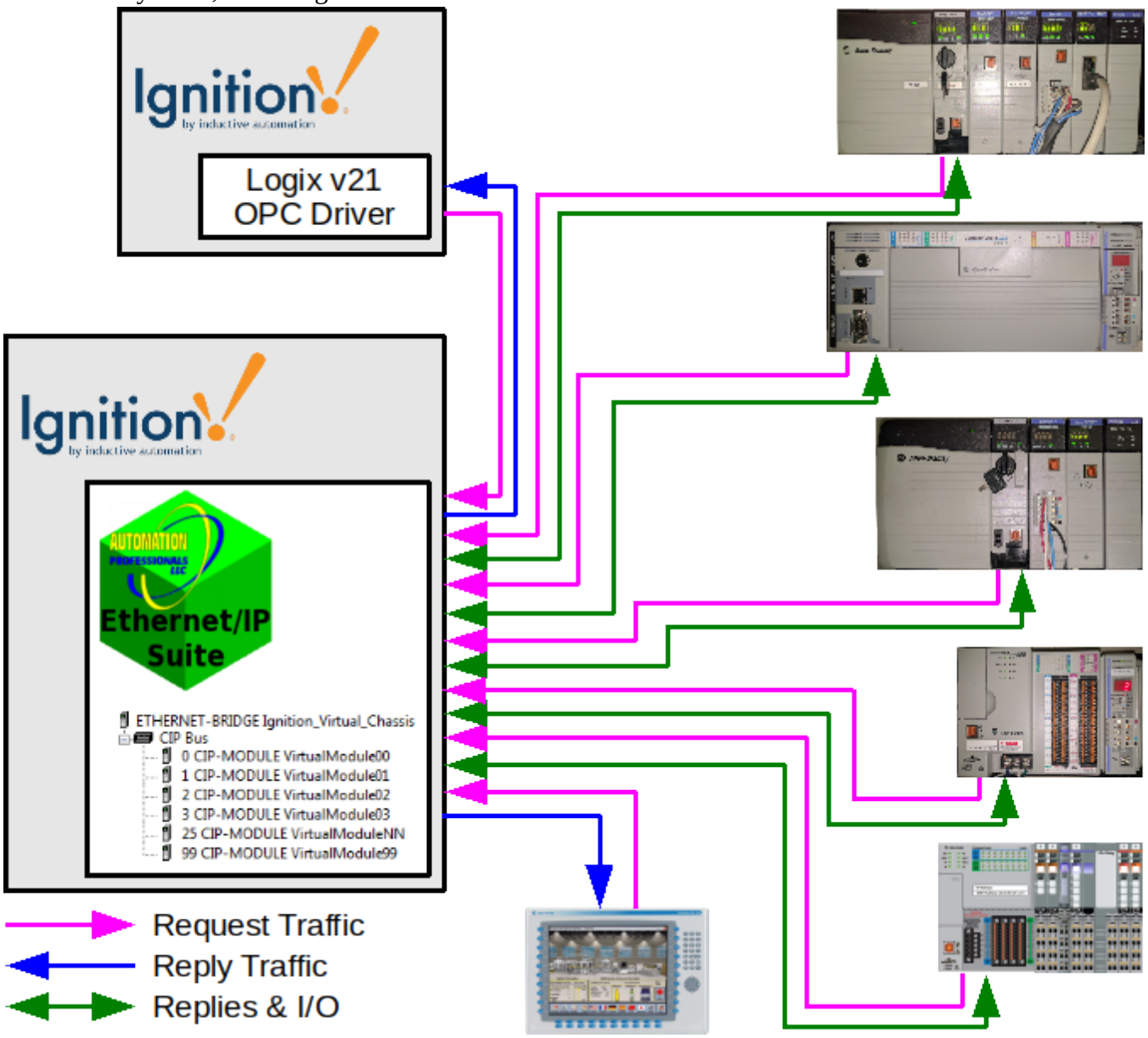


Figure 1: Generic Host Driver Network Layout

The Generic Host Driver, diagrammed in Figure 1, provides the following features:

- Implements a virtual CIP chassis allowing up to one hundred (100) driver instances per Ignition server, each in a specified slot. Although an IP address can only be attached to one driver instance, backplane message routing makes all instances reachable from any listening IP address.

- Emulates Logix data types and data tables (tags) as described in the Logix Data Access manual, accepting CIP data table read and data table write messages from Logix processors. Each driver instance has its own collection of data types and tag names, just like separate Logix processors in a single chassis. As a convenience, a real Logix processor's project file, exported in L5X format, may be imported to a driver instance to set up the same data types and tags, including initial values for most types. While not specified, this emulation is expected to be compatible with third party products' Logix drivers, including various industrial HMIs, and **is** compatible with Ignition's own Logix v21 driver. Where any tag or datatype does not conform to the requirements of Rockwell's data access manual, that tag is omitted from the processor browse.
- Implements user-configurable I/O targets (aka assemblies and connection points) that are compatible with Allen-Bradley's generic ETHERNET-MODULE and ETHERNET-BRIDGE devices in a Logix controller's I/O configuration. These assemblies may reference any other data in the driver instance as scatter/gather member items. While only one Logix processor may own an assembly as an output connection point in a driver instance, multiple processors may output to different assemblies in the same driver instance, and may input from any configured assembly.
- Exposes all tag data and most CIP object data in each driver instance to Ignition's OPC/UA server, with browsing support. Standard Logix tag names, subscripts, structure element names, and bit numbers may be used in OPC item paths, with special syntax for CIP object classes, instances, and attributes. Keywords are provided for accessing arrays of bytes as ASCIZ or UTF8 strings, and arrays of 16bit integers as UTF16 strings, along with support for standard Logix STRING data. Keywords are also provided to convert certain types to/from OPC DateTime values (timestamps).
- Allows all appropriately-sized emulated tags to produce to any Logix controller configured to consume them. No consumer or RPI limits are enforced, other than the performance of the Ignition server. Data types and tag sizes don't actually have to match—buffers will be silently truncated or zero-padded for the originator.
- Processes user-defined jython code for a variety of communication and data handling events, including events for OPC subscriptions. Each virtual device has its own code. These code modules are exposed via system.cip.* scripting functions for integration with gateway scripts, both shared and per-project. Ad-hoc CIP messages can be constructed and sent to the virtual modules and the replies retrieved, even from client/designer scope.
- Provides an "I/O Momentary Button" with reliable turn-off in case of Vision client UI failure or client-gateway communication disruption. It is designed to emulate physical Normally-Open or Normally-Closed pushbuttons attached to a Remote I/O chassis. This component bypasses the SQLTags infrastructure to set/unset the appropriate boolean in a virtual device's tag data structures.

Support for PLC-5 and SLC-500 messaging types is planned for a future release.

1.4 Enhanced Host Driver Features (Scanner)

The ***enhanced*** Host Driver, in addition to all of the generic features, adds ***originator*** functionality, that is, it can actively initiate communications with external devices. This includes both request/reply (scripted explicit messaging) and I/O type (scanner or tag consumer) connections.

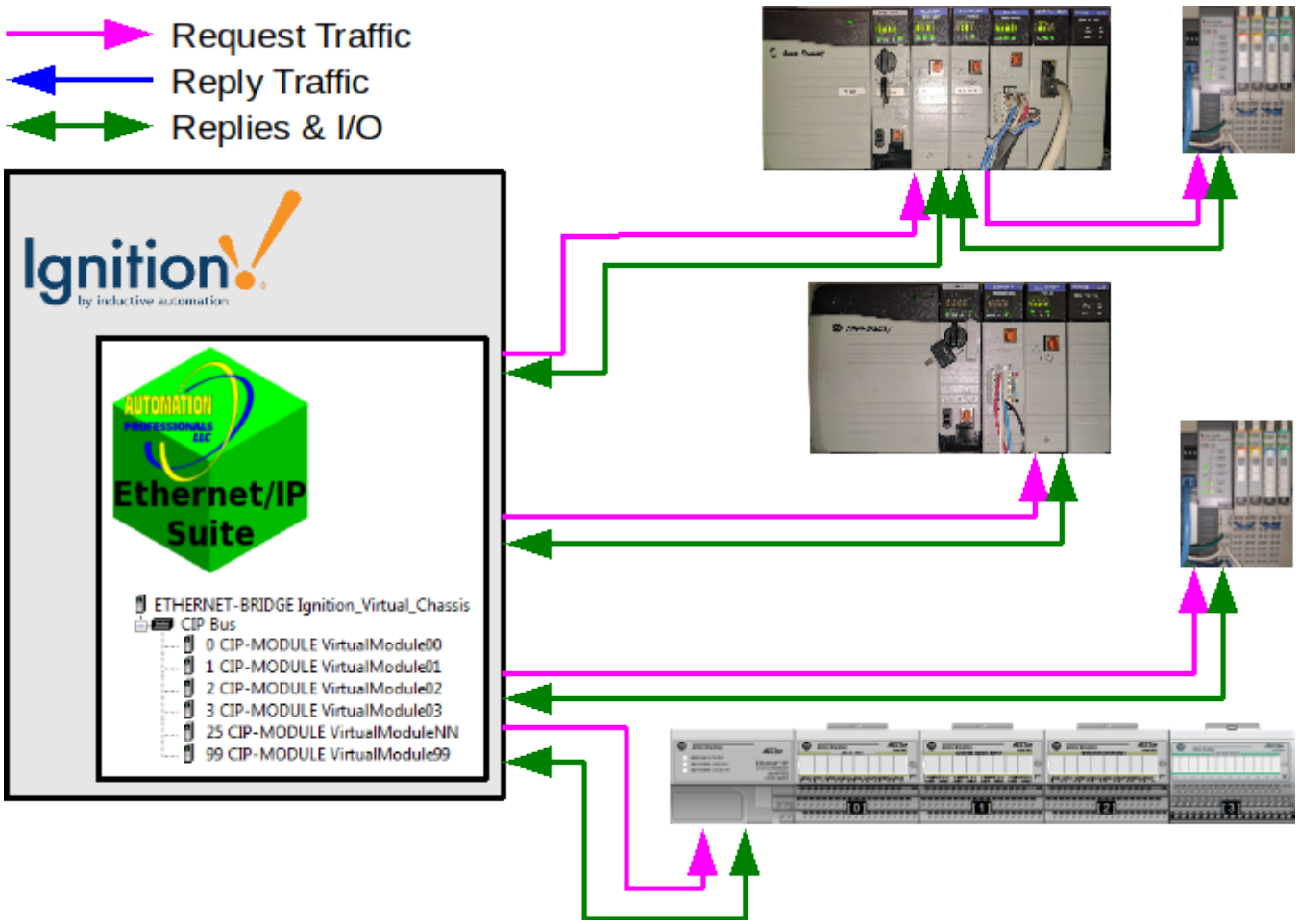


Figure 2: Enhanced Host Driver Additional Network Features

The Enhanced Host Driver ***adds*** specific features, as diagrammed in Figure 2:

- I/O module scanning via any configured local IP address, including support for connection paths through other EtherNet/IP bridges. (Local IP addresses must exist in the OS and be configured in a Host Driver to be used with scanner mode.)
- Support for consuming tags from Logix or other controllers that are configured to produce them. Data types do not need to match, but total tag size must match.
- An originator 32bit serial number for the driver, unique per slot, as required by the CIP specification. The module identifies itself with Automation Professionals' registered ODVA vendor number, 1311.
- Arbitrary CIP messages can be sent to and replies received from any external CIP device reachable from any configured local IP address.

1.5 PLC Feature Compatibility Matrix

	Ignition Client Features						Ignition Adapter Features			Ignition Scanner Features	
	Symbolic Tag R/W	Tag Instance R/W	Optimized Structure R/W	Large Structure R/W	Class/Inst/Attr R/W	Connection Buffer Size	I/O Connection Size	Consumer Tags (To PLC)	Unsolicited Messaging	I/O Connection Size	Producer Tags (From PLC)
Allen Bradley Logix ≤v20	✓	✗	✓	✓	✗	4000	500 x100	✓	✓	✗	✓
Allen Bradley Logix ≥v21	✓	✓	✓	✓	✗	4000	500 x100	✓	✓	✗	✓
Allen Bradley MicroLogix 8xx	✓	✗	✗	✗	?	4000	✗	✗	✓	✗	✗
Allen Bradley MicroLogix 8xxE	✓	✗	✗	✗	?	4000	500 x8	✗	✓	✗	✗
Omron NJ & NX1P2, Built-in Port	✓	✗	✓	✗	✓	1994	600 x32	?	✓	600 x32	?
Omron NJ with CJ1W-EIP21	✗	✗	✗	✗	?	✗	1444 x32	?	?	1444 x32	?
Omron NX102, per Built-in Port	✓	✗	✓	✗	✓	1994	600 x32	?	✓	600 x32	?
Omron NX701, per Built-in Port	✓	✗	✓	✗	✓	1994	1444 x256	?	✓	1444 x256	?
Keyence KV	✗	✗	✗	✗	✓	4000 (500)	1444 x256	?	✓	1444 x256	?
Automation Direct P1000-P3000	✗	✗	✗	✗	?	✗	1444 x4	✗	✗	1444 x4	✗
Generic EtherNet/IP I/O Adapters	✗	✗	✗	✗	✓	Varies	✗	✗	✗	Varies	✗

1.6 Firewall and Networking Requirements

1.6.1 Ports

The following Internet Protocol ports, reserved with IANA for the EtherNet/IP protocol, are required by the various options of this driver, as follows:

Port	Usage
44818/TCP	EtherNet/IP unencrypted message encapsulation. • Required outbound for all Generic Client driver connections, and for Scanner connection handshake. Also used outbound for fully scripted messaging via the Scanner option. • Required inbound for Adapter connection handshake and for Logix controller MSG target emulation.
44818/UDP	EtherNet/IP browse peer discovery and resource constrained messaging. • Used inbound and outbound by the Adapter option to expose Ignition to RSLinx. Optional.
2222/UDP	EtherNet/IP I/O data and producer/consumer data transport. • Required inbound and outbound for Adapter option and Scanner option functionality.
2221/TCP 2221/UDP	EtherNet/IP TLS and DTLS message encapsulation and encrypted I/O traffic. • Not currently implemented in this driver.

For full functionality, ensure your gateway's firewall allows traffic on these ports. In the inbound case, ensure the firewall permits the traffic at least on the IP addresses used by any EtherNet/IP Host device instance as Local Addresses.

1.6.2 Routes

EtherNet/IP messaging that travels over TCP/IP is generally compatible with all common routing topologies, including common forms of Network Address Translation that disguise the client application's real IP address from the target device. This means that:

- When using the Generic Client driver, Ignition's real address can be hidden (substituted) by source NAT when communicating with PLC target devices, and PLC target devices' real addresses can be hidden (substituted) by destination NAT (aka pinholes) behind virtual IP addresses.
- Similarly, when using the Logic controller MSG target emulation of the Adapter option of the driver, NAT in the other direction (when PLC is the client) is fine.

EtherNet/IP I/O traffic that is purely UDP Unicast cannot traverse any NAT. The handshaking to set up such connections is performed over a separate, ephemeral TCP channel, and embeds the real IPv4 addresses of each endpoint within the protocol content. Those real endpoints ***must*** be normally routable for UDP traffic ***in both directions***.

Furthermore, EtherNet/IP I/O that includes any multicast traffic (including producer/consumer tag functionality) not only ***cannot*** traverse any NAT, but is almost always confined to a common subnet. Many EtherNet/IP devices ***require*** multicast traffic in the target to originator direction (input data) as a consequence of redundancy support and/or support for listen-only connection types. If your application requires support for such traffic, Ignition ***must*** have an ethernet interface in the same layer 2 network, with an assigned IP address within the same subnet, as the peer device.

2 Client Driver

2.1 Settings

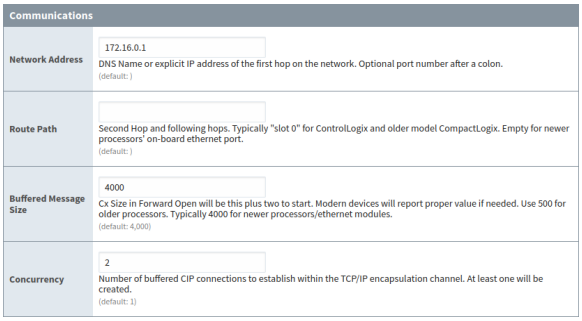


Figure 3: Client Driver Communications Settings

The "Communications" settings section provide the basic information needed to reach the target device, and tailor the use of concurrent buffered connections.

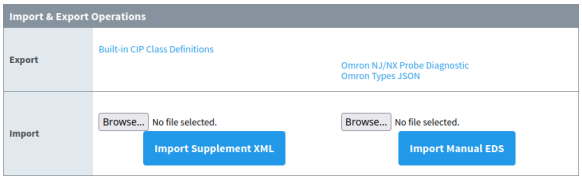
The "Route Path" setting configures the hops from the ethernet endpoint through any chassis or multiple chassis. It uses port segments, optional electronic key segments, and optional network segments. (But needing ekey or network segments would be very unusual.) The keywords are required. Leave entirely blank if the network port is on the front of the processor or intrinsically part of the device. (Use the first hop diagnostics in the OPC browser to verify the routing.)

The "Buffered Message Size" setting is the starting point for the driver's automatic adjustment routine. Very old devices may not cooperate with automatic adjustment—for those, use 500 (250 if DH+ is in the route path). The actual message size limit after adjustment is available as a diagnostic value. Once known, it can be placed here to avoid the few milliseconds needed for automatic adjustment.

The "Concurrency" setting is the number of buffered connections to use within the TCP/IP connection. At least one buffered connection will be used. These consume PLC resources even when idle, and can take time for the PLC to recycle, so do not use more than your application needs. Also check your target device's specifications for "connection resources".

There are a variety of advanced settings that are not detailed here. See the descriptions on each setting.

2.2 Configuration



After a driver instance is created with core settings, its Configuration page is used to access additional information and to provide supplemental CIP probe information and to provide an EDS file, if applicable.

If an XML supplement is imported, or a manual EDS file imported, links will be provided to re-export those files. The supplemental XML can add to or partially replace the information about CIP object classes, instances, and attributes that are built-in to the driver. Where "id" values clash, the supplement replaces the built-in. Otherwise, the supplement is cumulative.

If probing obtains an EDS file from the device, it will be exportable, too. If either Logix browsing or Omron browsing succeed, a gzipped raw probe file will be made available (for use by Automation Professionals' support), along with structured data type information and probed tag information in JSON format. The JSON-formatted exports may be used to create corresponding UDTs within an Ignition tag provider, and tags using those UDTs. Such UDTs can be used directly, or can serve as parent data types for inheritance, or can be altered as the end-user sees fit. UDTs for data types and Add-on-Instructions that have any members set to no external access should have those members pruned from the JSON before use. Also, in the compact form of these JSON exports, numeric arrays are defined as array tags, which may be more efficient, if a bit harder to use in many applications.

Some of the raw probe details are injected into the JSON UDT's documentation properties. Open the JSON in a syntax-highlighting text editor if you wish to review all such information in a convenient

form. Additional probe-related information is placed in the device's home folder within the gateway file system.

2.3 OPC Interface

The OPC Browser (and Gateway Quick Client) will show the user the tags discovered by probing under the `Controller:Global` and possibly also `Program:ProgramName` folders, organized hierarchically as one would find within a PLC's programming software. Within those folders, OPC item paths are strings conforming to "tagpath" syntax, as described here. Precise syntax allowed varies by target device—see the application notes by brand in the following section.

Information about the device connection and related meta-data is provided in the `[Diagnostics]` folder as usual, with some nodes that are specific to this driver.

Other folders may or may not be present, depending on the probe results, and OPC item paths within those folders may be more complicated Application Path strings, as described here.

When the target is a Keyence KV PLC, its linear address ranges will be shown under the top-level `Keyence Device Memory` folder. These address ranges are broken into many virtual subfolders to minimize the scrolling necessary to find any particular item.

Especially note the features invoked by special symbols in application paths, described in Special Symbol Segments.

3 Client Driver Application Notes

When the client driver starts up, before starting the network connection, it extracts the information from the last successful Logix and/or Omron raw probe files, and any previously extracted EDS file. If the target device is unchanged, this enables faster startup of existing subscription items, as they won't have to wait for the connection probe to complete.

The driver performs a complete probe after the connection succeeds, and again after any connection breakage/reconnection events. While connected, this driver will check Logix or Omron metadata once per minute to determine if a reprobe is required. If required, any reprobe will be conducted in parallel with other OPC services.

3.1 Allen-Bradley ControlLogix & CompactLogix

These processors have publicly documented CIP services to browse the available tags, browse the referenced structure types, read meta-data about the program version, and perform read and write operations upon discovered tags. Browsing includes program tags.

The public documentation notes that, starting with firmware version 18, tags and structure members may have external access controls applied. While there is partially documented metadata that indicates that specific tags are read-only (and tags with no external access are simply not listed in a browse), there is no corresponding metadata that indicates what a structure member's access rules happen to be. That information can only be obtained by actually reading or writing the member or its containing structure. While operating, this driver will annotate its internal probe information with any permissions failures it encounters on complete structures, so that later requests will go directly to the structure members instead. Note that this fallback operation greatly limits optimization of the tag in question. Structure members that are not readable at all will be individually attempted on every cycle, so should always be pruned from Ignition's tag list.

In order for this driver to optimize reads of structure contents (by reading entire structures at once), ***all*** of the user-defined types' members must have at least "read only" external access. This driver will attempt to read complete structures any time ***two*** of the structure's members are included in the same

subscription pace, or in the same multiple-item OPC read. This means that you should always use the same subscription pace for members of the same structure, unless there's a no-access structure member that prevents optimization.

In order for this driver to optimize writes to complete structures, ***all*** of the user-defined types' members must have "read/write" external access, and ***all*** of the data for ***all*** of its members must be submitted in a single multi-item OPC write operation. (Where named booleans are hosted by another member, which is often hidden, only the named booleans must be written. Remaining bits will be zeroed, and are expected to be ignored in the PLC.) When writes to structures cannot be optimized, individual member writes will occur.

An alternate optimization occurs when working with individual bits of a single integral type instance, including named booleans sharing a single host member of a structure: multiple bit reads will be combined into a read of the enclosing integral type, and the bits extracted in the driver. Similarly, multiple bit writes will be combined into a masked read-modify-write service altering all of the given bits in one operation.

The easiest way to correct the external access problem is to bulk replace the definitions via a Studio5000 export/import operation:

- Export your Studio5000 project in L5X format, making sure the "Encode Protected Content" checkbox in the lower left is **not checked**. (You may need to adjust your options if that checkbox is checked and greyed-out.)
- Open the exported project in a text editor, and position the cursor on the line containing `</AddOnInstructionDefinitions>` (verbatim). This will be right before the first section of `<Tags>`.
- Select all of the text from there to the **beginning** of the file (Ctrl-Shift-Home in many editors).
- While that region is selected, open the editor's Search & Replace tool, and set it to "plain text" and "limit to selection". (Terminology varies.)
- Replace all cases of `ExternalAccess="None"` with `ExternalAccess="Read Only"` in the region. Save the file and exit.
- Open the modified L5X in Studio5000, which will prompt you to create a new ACD file. Download this updated file to your PLC.

3.1.1 Predefined Data Types

The public documentation instructs 3rd parties to avoid accessing complete structures of these datatypes. In many cases, this is required as a consequence of external access limitations on members of these types. This driver will optimize where it can. Since these types are predefined, there is no work-around possible.

If a predefined type has any unreadable member (many do have hidden members, especially the ones for motion and for function block instructions), including instances inside of other data types "poisons" the outer data type, making it also not readable as a whole.

3.1.2 I/O Module Data Types

The public documentation instructs 3rd parties to avoid accessing complete structures of these datatypes. Like predefined types, external access limitations can prevent optimization. More importantly, I/O modules can use structures that don't obey the normal Logix data alignment rules, and can include bit fields as fragments of integral types. This driver will optimize where it can. Since these types are I/O module defined, there is no work-around possible.

Like the processor's pre-defined types, I/O module types often have unreadable members. Do not include such types inside of other user-defined types.

3.1.3 Add-on Instructions

Again, the public documentation instructs 3rd parties to avoid accessing complete structures of these datatypes. Certain members of AOI types are hidden or, in older firmware, simply implied. The observed behavior is not too complicated, so this driver will optimize where it can. Access control is a potential problem. However, since these types are usually user-defined, there ***are*** work-arounds available.

Note that ***no*** external access ***is the default*** for local tags within AOI definitions. These must be changed in Logix 5000 or Studio 5000 to at least "read only" external access. Be aware that the external access column is ***hidden by default*** on the local tags definition tab within Rockwell's programming software. The column must be manually added to the display before one can fix the external access setting of these tags. An AOI definition's `enableIn` and `enableOut` boolean tags are also locked to "read only" external access, so this driver can never optimize ***writes*** to complete AOI structure tags.

If you've purchased a processor with AOIs built into its firmware, and the AOIs have any non-readable members, you will not be able to make the changes needed to enable optimized read access.

If you've obtained an AOI from a third party, and it is sealed, it will still be OK for optimization if the third party prepared it with all parameters and local tags set to at least Read Only. Contact that third party and request an updated AOI if you encounter a problem.

As mentioned above, including any pre-defined or I/O module-defined data type that has non-readable members in an Add-on Instruction will "poison" the AOI's data type, preventing optimization. In particular, AOIs should not include any function block diagram routines, as many of their instructions have this problem. As a work-around, define any such troublesome items as In/Out parameters, which do not impact the rest of the AOI's data type definition.

3.1.4 Structure Alignment Rules

{ Also see the in-depth review in Structure Definition Segments. }

Prior to firmware version 27, all Logix user-defined and add-on defined data structures used 32-bit member alignment for everything, except for single primitive values of smaller data types. This means all structures are padded to multiples of 32 bits, all nested arrays and structures are aligned to 32-bit boundaries, and 64-bit data types are also aligned to 32-bit boundaries.

Beginning with firmware version 27, user-defined and add-on defined data structures that contain any 64-bit primitive values, directly or in nested structures, use 64-bit alignment instead. The 64-bit alignment applies to everything, except for single primitive values of smaller data types, but does not affect the internal alignment or size of nested 32-bit structures.

This driver does not attempt to determine the alignment used in a Logix data type, and does not use the target device's firmware version, but simply uses extra padding members where necessary to achieve the overall structure size and individual member offsets reported in the browse process. Including extra members in a structure is part of the algorithm to support AOIs in older firmware, so that functionality is repurposed for this case. But care must be taken if the resulting structure is exported for use in the Host Driver (via the XML placed into the device's home folder).

3.2 Allen-Bradley MicroLogix 800 family

These processors have public documentation for messaging applications, which covers much of the requirements for this driver. While only indicated obliquely in this documentation, these processors

partially implement the services described in the Logix Data Access manual for ControlLogix and CompactLogix processors. The processor family has sophisticated data structure support, and support for arrays where subscripts are not zero-based. However, they do not report structure information, and there is no way in the Logix Data Access browse process to expose the array subscript ranges. To use this driver to directly access ML800 family tags, without having to set up Modbus mappings, you must:

- Use primitive data types—no structures.
- Use ML800 native strings, no ControlLogix/CompactLogix style string structures. (ML800 native string are actually CIP `Short_STRING` instances with a defined maximum length.)
- Use zero-based arrays.

Any tags that do not follow these rules will simply not be reported by the PLC during the driver's probe.

These processors do not support the CIP Specification's Message Router Multiple Request Service, that permitting batching of many requests within a large request buffer. The driver will automatically detect this lack of support, but that short delay can be preempted with an advanced driver setting.

3.3 Omron NJ/NX family

These processors have public documention for their data types and related information for use in messaging applications, but do not document their browse process for tag and structure information.

The browse process has been reversed engineered in considerable detail, including identifying arrays with non-zero-based dimensions, multi-dimensional array tags and structure members, all documented primitive data types, and the layout of structures of type NJ and type CJ. Unfortunately, the browsed information for structures does not explicitly identify which layout is used. It must be inferred.

3.3.1 Structure Alignment Rules

{ Also see the in-depth discussion in Structure Definition Segments. }

Structures defined with the preferred "NJ" rule have variable alignment, based on the alignment requirements of their members. The linked documentation, in §A-5-1, specifies the alignment requirements of all of the PLC's native data types. Structures of type "NJ" align each member according to its own requirement, pad the structure size to match the largest alignment requirement among its members, and use that largest alignment value as the structure's own alignment if nested in another type. Booleans are not packed together in "NJ" data types.

Structures defined with the legacy "CJ" rule have fixed 16-bit alignment for all members, even individual bytes, and strings (which otherwise would use alignment=1). But consecutive boolean members are packed together into one or more (nameless) 16-bit WORDs. Data types, including non-CJ structures, are forced to align to 16-bit boundaries instead of their normal alignment.

After probing an NJ/NX processor, the algorithm lays out the members using the NJ rule, and if the resulting length is correct, it uses it. Otherwise, it lays out the members using the CJ rule, and if the resulting length is correct, it uses it. Finally, if neither length matches, a warning is logged. In this case, the NJ layout is used, but marked to not be optimized. This causes every member to be individually accessed, avoiding the layout error.

3.3.2 Structures with User-defined Member Offsets

Omron's Sysmac Studio permits the creation of data types where each member is given an explicit byte offset, or for booleans, an explicit byte and bit offset. Sysmac Studio does not permit members to overlap, and appears to pad the entire structure to a 16-bit boundary.

It is possible (easy, even) to define a structure with user offsets that produces a length match to one of the normal rules, but with a different actual layout. This cannot be detected by the driver, and should be validated manually for any application that has user-defined offsets in any data types.

Whether a warning is generated or not, a custom data type should be prepared that actually matches the true layout, and supplied to the driver in the Supplemental XML file, with an exact name match. When the probe algorithm acquires all of the raw member information from the PLC, before laying them out, it will attempt to substitute the supplied layout. The supplied structure's length must match, and every member named in the PLC must be present in the substitute.

3.3.3 Omron Strings

Strings in these Omron processors are stored as a simple null-terminated byte array, where the byte array is pre-allocated as a fixed size. The browse procedure does supply this fixed size information. As noted in the public documentation §A-5-1, these are 8-bit aligned. However, this conflicts with the special handling rules for arrays of strings described in §8-7-4.

Based on reverse engineering results, strings and arrays of strings embedded in structures are actually 8-bit aligned, and can be read or written in bulk when the entire structure is read or written, and are ***not*** byte-swapped in that case. In addition to that discrepancy, attempts to perform multi-element array reads as described in §8-7-4 produced error responses from the PLC, and repeated attempts broke the entire connection.

As a consequence, any optimization result that *would* result in a multi-element string array access, is preemptively replaced with multiple individual reads. If optimized string array reads are needed, make the array part of a structure, include at least one other member of the structure in the PLC, and include that extra element in your subscription or bulk read operation.

3.3.4 Omron Booleans

The NJ/NX family of processors default to storage of booleans in 16-bit words, when used as individual tags, and when used as individual members of the default "NJ" structure format. When placed in "CJ" format structures, consecutive named booleans will be packed into the 16-bit words instead of taking one whole word each.

When arrays of booleans are created, in either case, the individual booleans are packed into as many 16-bit words as needed, with multiple dimensions packed closely together.

When booleans are accessed on the wire, as booleans, individual booleans are transferred in a 16-bit word. Arrays of booleans are transferred using one byte per boolean (Not Documented!). This is true whether the booleans or boolean arrays are individual tags or members of structures.

However, when an entire structure is accessed, any booleans within are transferred in the packed format, if applicable, that the processor uses internally. Because of this, booleans in these processors should always be placed in structures, preferably CJ-formatted structures, along with at least one non-boolean member. The non-boolean member should be read/subscribed with the booleans to trigger optimization at the structure level.

3.4 CIP Attribute Access

All EtherNet/IP devices are required to support a variety of addressable attributes, like Identity information, and usually support additional functionality, like TCP/IP configuration attributes and information about uploadable EDS files. The most commonly used classes from the specification have data types predefined in this driver for their attributes. Additional data type definitions may be added by the user for application-specific uses (like structured assembly data).

3.4.1 Simple Attributes

The majority of attributes defined in the CIP Specification are single instances of an elementary data type. The application path to use in the OPC Item Path is simply the class, instance, and attribute ID information, possibly with macro shortcuts. Something like this:

```
cls C inst I attr A
```

Substitute actual integers in place of the C, I, and A placeholders. The value will be read using the standard CIP service code 14 “Get Attribute Single” and written with service code 16 “Set Attribute Single”. The values on the wire will be converted to and from OPC standard data types as long as the attribute is listed in the built-in CIP types, or is defined using the supplemental XML file. If not defined, the attribute will be read as a byte array, and a byte array must be supplied when writing.

Certain attributes in the specification use the STRINGI data type. While listed with the elementary data types, it is composed, potentially, of several language-dependent strings, and needs to have a three-letter language code supplied to access its content.

3.4.2 Complex Attributes

The CIP Specification does define some attributes as structures, arrays, or arrays of structures. They must still be read or written in single requests, but this driver can extract specific elements or assemble a write from a batch of simple values. Use additional selectors after the attribute code to drill down to an elementary datatype. Use parenthesis around the tokens if the following selectors are in tagpath format. Something like this:

```
(cls 55 inst 0 attr 32)[0].Name.eng
```

That class attribute of the File object happens to be the directory, and is a variable-length array of structures, each containing an instance number and two STRINGI values. That complete application path selects the english language form of the first structure’s Name member.

When multiple items in a batch read or subscription select values from a single attribute, they will be optimized into a single read request. Writes to such complex attributes must supply all elements in a single batch, as the entire attribute must be written in a single request.

The driver support for complex attributes depends on the built-in definitions from the CIP Specification, and any user-defined types and attribute definitions in the supplemental XML file. Complex attributes that have a variable-length array in their definition cannot be written in complex form. Use a byte array of the encoded content to write such attributes.

3.4.3 OPC Browsing of Attributes

Some classes’ instance zero attributes will be exposed during the driver’s initial device probe after connection, and will be immediately browsable. Otherwise, nodes for attributes will be created on first OPC request, and will become browsable thereafter. This means that most OPC Item Paths for these attributes must be manually entered in an OPC tag, or dynamically generated in an Ignition UDT, or requested with a `system.opc.read*()` script call. The support for browsability is intended to simplify troubleshooting with the OPC Quick Client, not for dragging and dropping to create Ignition tags.

3.4.4 Polling I/O Assemblies

Note that assemblies in devices can be polled/written with this method. Out of the box, you will get byte arrays on read and must use byte arrays to write. But you can use the supplemental XML configuration file to define the actual structures involved, and assign those structure types to the assembly instance’s data attribute. (Which is always attribute #3.)

Imagine an EtherNet/IP analog input module that reports its status in assembly #101, with some status bits for channel underrange, overrange, and an overall module OK bit, followed by 16-bit integers for the raw sample data. You could supply a supplemental XML like this:

```
<?xml version="1.0" encoding="utf-8" ?>
<cip>
  <struct name="Input_Buffer">
    Status1=INT
    Ch0_Under=HostedBy Status1.0
    Ch1_Under=HostedBy Status1.1
    Ch2_Under=HostedBy Status1.2
    Ch3_Under=HostedBy Status1.3
    Ch4_Under=HostedBy Status1.4
    Ch5_Under=HostedBy Status1.5
    Ch6_Under=HostedBy Status1.6
    Ch7_Under=HostedBy Status1.7
    Module_OK=HostedBy Status1.15
    Status2=INT
    Ch0_Over=HostedBy Status2.0
    Ch1_Over=HostedBy Status2.1
    Ch2_Over=HostedBy Status2.2
    Ch3_Over=HostedBy Status2.3
    Ch4_Over=HostedBy Status2.4
    Ch5_Over=HostedBy Status2.5
    Ch6_Over=HostedBy Status2.6
    Ch7_Over=HostedBy Status2.7
    Ch0=INT
    Ch1=INT
    Ch2=INT
    Ch3=INT
    Ch4=INT
    Ch5=INT
    Ch6=INT
    Ch7=INT
  </struct>
  <class id="4" name="Assembly">
    <instance id="101" name="Some Analog Inputs">
      <attribute id="3" name="Data" type="Input_Buffer"/>
    </instance>
  </class>
</cip>
```

Then you could access all of these elements by name, using the macro for assemblies:

```
(assy 101 attr 3)Ch0_Under
(assy 101 attr 3)Ch0_Over
(assy 101 attr 3)Ch0
```

As long as these OPC items are subscribed at the same rate, a single request on the wire will satisfy them all.

3.5 Keyence KV family

Keyence PLCs that support EtherNet/IP via their built-in ports, or via either the KV-EP21V or KV-NC1EP communication add-on module, are recognized during the driver startup probe, and the PLC model code is read to determine the presence and max sizes of the supported device memory areas. For the KV-8000 and KV-8000A models, the function version is also read from Control Memory #900 to determine the correct range for “R” relays.

3.5.1 OPC Browsing

The **possible** addressable items are exposed in the folder “Keyence Device Memory”, organized by address type. Virtual folders are presented in a hierarchy to limit the amount of scrolling required to browse to specific addresses when thousands of addresses are present. See Keyence Application Paths for the manual addresssing syntax. The user is responsible for limiting themselves to the **configured** global address range in the target PLC, for each address type. (Most address types’ ranges are split between global and local variables.) Access to addresses in the **local** ranges will be rejected by the PLC.

3.5.2 Communication Buffer Optimization

These PLCs allow 4000-byte communication buffers, but cannot actually use 4000 bytes in any messages. To minimize resource usage in the PLC, set the buffer size to 500 bytes. (The largest possible message is ~460 bytes.)

3.5.3 Request Optimization

These PLCs do not allow multiple small requests to be consolidated into a larger request using the EtherNet/IP standard “Multiple Request” service code. Therefore, the only optimization possible in this driver is to read consecutive addresses wherever possible. Gap spanning is always used to request message quantities, except for Control Memories and Control Relays, as those are the only address types that forbid access to undefined locations. This means that overall driver performance will be greatest if PLC memory is allocated for tags in consecutive addresses.

4 Host Driver

4.1 Settings

Each driver instance functions as a virtual I/O module and virtual Logix processor in one of the 100 allowed slots on the virtual backplane. Like all Ignition devices, there is a "General" settings section for Name and Description, and whether the device is enabled or not.

The "Security" settings section offers the option to restrict some functionality based upon authenticated roles. When no roles are listed, no restrictions are applied.

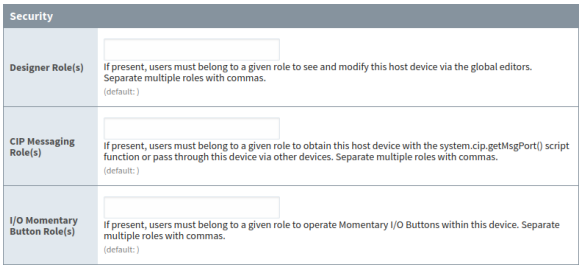


Figure 4: Host Driver Security Settings

The "Communications" settings section defines the behavior of the Port Manager shown in Figure 21 in the Host and Target Drivers' Functional Description section. At a minimum, you must supply a unique slot number.

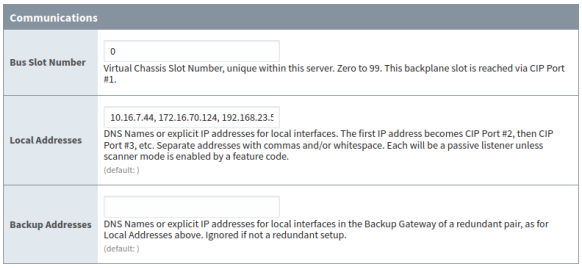


Figure 5: Host Driver Communications Settings

4.1.1 Local Listening IP Addresses

In a Host Device, the "Local Addresses" setting is responsible for creating the "EtherNet/IP Port #2" and above shown in the Port Manager of Figure 21. The IP addresses supplied here must be **local** IP addresses of the server. These are **NOT** the addresses of external devices. Only one driver instance can claim any specific IP address. Use backplane slot addressing to reach other driver instances via the slot that has the IP address assigned.

DNS host names may be used instead of IP addresses in this setting, but they must resolve to the real IP addresses of the gateway when the device is started.

Note: only IPv4 addresses are supported, due to limitations in the EtherNet/IP specification itself. Also note that NAT (network address translation) between gateway and remote devices is also not supported, in either direction. (Different subnets are supported if both directions use unicast.)

4.1.2 Backup Listening IP Addresses

In a redundant pair of Ignition Gateways, the listening addresses must be different. The master gateway will use the addresses above, in the Local Addresses field. The backup gateway will use the addresses in this setting, which must have the same number of entries as Local Addresses. Ignored if not a redundant setup.

If DNS host names are used above, and they resolve to the backup server's real IP addresses when the backup server looks them up, this setting can be blank.

4.2 Configuration

After a driver instance is created with core settings, its Configuration page is used to set up the rest. The balance of the configuration contains the data types and virtual tags the Logix emulation offers, the assembly connection points and their scatter/gather definitions, and, for the Enhanced Host Driver, the I/O modules to scan. In contains all of the data values the module should start with. It is maintained internally as XML.

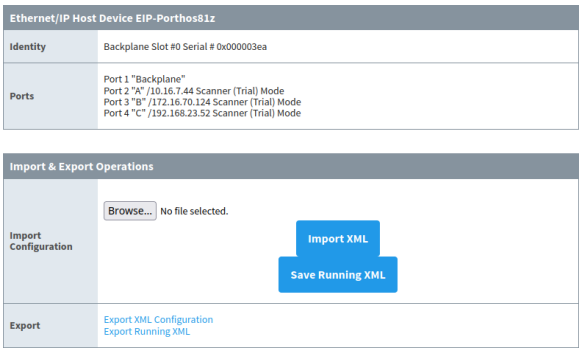


Figure 6: Host Device Configuration Import/Export

When disabled, only the summary data and the XML import/export section are available on the configuration page, as shown in Figure 6. The summary data displays the Port Manager details that are in effect from the settings and from the module license, if any. (A default serial number is used when no license is present.)

4.2.1 Default Configuration

When either a Host or Target device is first created, it is populated with sample data types, tags, and assemblies that allow each to immediately communicate with an external PLC's I/O scanner as a generic device. The corresponding module settings for an Allen-Bradley Logix processor are shown in Figure 7.

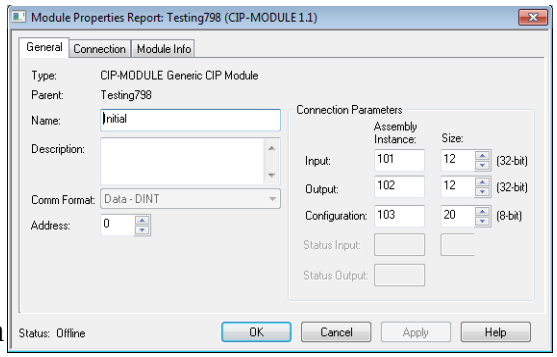


Figure 7: Sample Configuration Connection

The first 8 bytes of output data are echoed back to the PLC's input data without change. The balance of the input data block is populated with counters and bit patterns derived from them. See the Jython code included in the sample configuration for details on the generated data.

The sample configuration for a Host Device also contains a scanner module suitable for use with another server's sample device, as an example of Active I/O Scanning. Adjust its target IP address and enable it if you wish to experiment.

4.2.2 Configuration XML Format

The XML format described above is based on a simplified form of the Logix L5X file format, with additional sections for non-Logix features. The "Import XML" operation accepts both the condensed format or an existing Logix Processor's .L5X files (without *encoded* Add-On-Instructions). The import process will identify data types, controller tags, and I/O modules, and reduce the XML to just the supported functionality.

Unfortunately, the L5X file format does not include all necessary details about its I/O modules. After importing from an L5X, the EDS files for the I/O devices will be needed to create working I/O slaves. See the section on Interpreting EDS Files below.

The Import XML operation ***completely replaces*** the previous configuration. If a merged configuration is needed, cut-and-paste the necessary XML pieces together. The condensed XML format is organized to be much more readable than an L5X, to help with cutting and pasting.

4.2.3 Runtime Configuration

When a “Host Device” instance is enabled in the settings page, additional configuration page sections are displayed. All of these additional sections display and offer editing of the ***running driver instance***. Changes take immediate effect in the server, but are ***not saved*** in the startup configuration. When all of the configuration items are working as desired, and data values suit, use the “Save Running XML” button to create an XML snapshot and place it in the internal database. This is equivalent to using the “Export Running XML” link to save to a file, then importing that file.

4.2.4 Data Type Emulation

The first runtime configuration section lists all user-defined and I/O defined data types used throughout the rest of the configuration. System-defined datatypes corresponding to a LogixV20 processor are preloaded into the emulation and are not displayed. See the “Lgx Type Manager” class object’s instance list in the OPC browser or Quick Client for more details on the pre-defined types.

While a Logix processor keeps track of the difference between User-defined data types, types from Add-On Instructions, and Module-defined data types, they are exposed to the world in one big list of definitions, so this emulation treats them the same. When parsing an L5X that has I/O or AOI data types, they are added to the user-defined list within the emulation, and become fully editable. (Note: an L5X file does not always contain complete information about I/O data types—manually check for alignment and bit placement.)

The data type editor within the gateway configuration editor uses the syntax described for members of anonymous structures described in §8.4.4 below. However, for your convenience when working from Logix L5K files, accepts that same syntax for member fields, as long as you prune the member description, visibility, and access control information. Therefore, each line defining a member may have one of these forms:

1) *membername=definition*
2) *datatype membername*
3) *datatype membername* [*dimension*]
4) BIT *membername hostmembername* : *bitnumber*

Form #1 accepts the full range of definition options described in §8.4.4.

The data type in forms #2 or #3 must be one of the elementary types, a predefined structure type, or a different datatype that doesn’t depend on the current type (no circular references).

The hostmembername in form #4 must be an intrinsic type elsewhere in the same member list, and the bit number must fit within it.

Alignment is specified by drop-down selection, and would typically be “32bit DWord (Logix)”. The system will preserve the case of member and datatype names but is not otherwise case-sensitive. These names must follow Logix rules for identifiers, basically allowing just alphanumerics, underscores, and colons. To avoid possible compatibility problems, only use colons in I/O type names. Consider replacing colons with underscores to improve scripting support.

Use “64bit LWord (Logix)” alignment if Logix Firmware v27+ compatibility is needed, and any member of the structure is a 64-bit type, or has a nested 64-bit type.

4.2.5 Tag Emulation

The second runtime configuration section lists all virtual controller tags and their data types. Tags have a name, a data type, and optional dimensions. No limit is placed on the dimensions, other than general

Java memory consumption. If a tag's array dimensions are changed, or a structure data type is changed while being used by a tag, the tag will be reconstructed and an attempt will be made to retain its content.

Like Type Names, Tag names must follow Logix rules for identifiers, basically allowing just alphanumerics, underscores, and colons. To avoid possible compatibility problems, only use colons in I/O tag names. Tags receive special treatment in a Logix Processor's Message Router, where the tag may be specified by name in an ANSI Symbol segment instead of by class and instance. The driver fully implements the corresponding CIP services, and tags are listed by name in the OPC browser in the root of the driver instance (along with the CIP Object Classes).

If the license permits I/O scanning, tag consumption from Logix controllers is also allowed. Specify the route path to the controller through the appropriate outbound port, name the tag to consume, and specify the packet pace desired. These fields are ignored if scanner functionality is not available.

4.2.6 CIP Assembly Emulation

The next runtime configuration section lists the assembly connections points defined for passive I/O targets, with a summary of the data it refers to.

Support for the generic Logix `ETHERNET-MODULE` and `CIP-MODULE` within the generic Logix `ETHERNET-BRIDGE` chassis type require the use of these assembly numbers, also known as connection points. Within this driver, an assembly must point to one or more tags or elements of tags. The assembly itself stores no generic data. The data in the driver instance does not have to match the data type chosen in the Logix module definition. The assembly emulation doesn't dictate any restrictions on the RPI used by the Logix Controller, although garbage collection within Java may interfere with very fast packet rates.

While intended to emulate the generic Logix Ethernet I/O modules, the driver instance can define any assembly connection point needed to emulate other I/O modules, too. These assemblies can also be targets of other Ignition Servers running this module with Scanner features enabled. Support for standard CIP Parameter objects and Parameter Groups is planned for a future release.

Assemblies distribute multiple data values from a single receive buffer and combine multiple data values into a single send buffer. These operations are typically called scatter and gather, respectively. The CIP standard for assemblies specifies scatter/gather operations down to individual bits. Assemblies have a name (for convenience only), a connection point number, and a list of members. Although connection point numbers in the CIP specification can be 32-bit numbers, Logix processors can only use 8-bit and 16-bit connection points.

Each member specifies a number of bits and a data object's application path. The number of bits used does not have to match the encoded size of the data object. The data object must support the read or write operations the assembly will be used with. The driver does have one specific limitation: members must either be byte aligned, or the necessary bit shifting operations must fit within a 32-bit temporary register. The simplest assemblies have a single tag in their member list, deferring all packing and alignment to the data type of the tag.

As of version 1.4.x, an Assembly's member list can include padding members of a specified number of bits. Simply include the bit width of the padding member on a line by itself when editing.

Also, as of version 1.4.x, Assembly member items' application paths may begin with a single "slot n" to reference an application path in a *different driver instance*. If the rest of the path is in tagpath format, remember to use parentheses around the slot tokens.

Assemblies may include structure tags that use any desired alignment rule, simplifying some cross-brand connections.

4.2.7 I/O Module Scanning

The final runtime configuration section lists the I/O modules defined in the I/O Scanning Manager. This section needs a feature code to function beyond trial mode. Each I/O module definition supplies the following information:

- Name and Identification of the target for Electronic Keying, optionally including a catalog number for reference.
- Connection detail, including the route path to the target device, whether the connection is inhibited or enabled, and the mode and pace information in each direction.
- Data detail, including the application path within the target device, and the corresponding data objects in this driver instance. If there is a static data item in the application path, and dynamic config data specified with a config tag, the two are concatenated before use.

The route path starts with a port number/address segment for the outgoing EtherNet/IP port and the first hop IP address. Importing an L5X will typically get all but the beginning of this field correct. Simply replace “slot n port 2 ipaddress” with “port m ipaddress”, where “m” is the correct outgoing port number (typically ‘2’).

After importing an L5X with Logix generic I/O devices, the rest of the imported parameters for them are likely to be correct. For other I/O devices in an L5X, at least the application path will have to be constructed with the help of an EDS file or a Wireshark packet capture. Since EtherNet/IP I/O devices are not required to follow Logix UDT alignment rules, any imported I/O data types may need to be adjusted or split into multiple tags.

Use an assembly as an intermediate buffer to perform scatter/gather with unaligned/misaligned data types. As of version 1.4.x, assemblies used by the scanner can include bit padding and can scatter to/gather from nested UDT elements.

5 Target Driver

The Target Driver is an abbreviated instance of the Host Driver, omitting all Scanner functionality and live editing features, but allowing a single configuration to occupy multiple slots in the virtual chassis.

5.1 Settings

This is similar to the Host Driver, but without any configuration options for Local Addresses or Backup Addresses. Where the Host Driver has a field for its Bus Slot Number, the Target Driver instead has a field for a comma-separated list of slot numbers and/or hyphenated slot ranges.

5.2 Configuration

The Target Driver uses the same XML configuration format at the Host Driver, but ignores any scanner module definitions. It offers import, export, and live export functionality on its configuration page. OPC Item Paths for configured virtual tags are the same as for the Host Driver, but with a slot segment prefix. An XML file created and edited in a “Host Device” can be loaded in a “Target Device” (unsupported features will be ignored).

In addition to the tags specified in the configuration, the driver pre-defines two read-only tags:

- `_slot` is an integer corresponding to the actual occupied slot in the virtual chassis, and
- `_relslot` is as above, but relative to the first configured slot for the target device.

These tags may be used as dynamic subscripts in OPC Item Paths and in Assembly Member definition’s Target attributes. See the example in §6.1.2 for how this can be used to consolidate assembly data from many virtual adapters into arrays in a host driver.

Note that the live export option only provides the live data from the first slot given in the settings. Similarly, the “Save Running XML” button will save the live data from just the first specified slot to be the startup data for all slots. If unique startup data is needed for each slot, use the jython code block to perform the customization (conditioned on the built-in `_slot` or `_relslot` tag).

6 Host and Target Driver Application Notes

6.1 Using Passive Target I/O

Allowing an external Logix controller, Omron controller, or other controller with EtherNet/IP scanner support, to initiate the I/O data traffic offers a number of benefits to the architecture of a control system:

- User interface objects that replace physical control panel buttons, lights, and similar components, can reliably operate using the same logic as those physical devices, and with similar user interface latencies. From the scanning controller's perspective, they are indistinguishable from real devices. Communication status is reflected in the controller using the same module status words that would condition inputs from a physical control panel. A programmer or technician can even use I/O forces in the controller when troubleshooting. Be sure to use this module's "I/O Momentary Button" for highly reliable momentary pushbuttons.
- Laying out tags in controller memory for efficient access from an OPC driver is no longer a design consideration on the OPC side. Data for an I/O connection is naturally packed into single packets, typically up to 500 bytes in each direction (larger with some PLC brands/models), with near-zero per-item overhead. Multiple emulated I/O modules can be used to optimize update rates, via the scanning controller's RPI settings.
- I/O data is transmitted with fire-and-forget UDP packets. This allows fast packet rates without risking the hiccups and hesitations that occur in TCP connections when packets are lost in transit. High resolution data can be transmitted in each packet in a best-effort recording scheme, or with simple echo algorithms for burst-transmitting buffered records.

6.1.1 Solo I/O Module Emulation with Logix Processors

For small data transfer applications that only need a single pair of 500-byte buffers, targeting a single driver instance, the Generic Ethernet Module is the easiest to configure. In full-size 1756-Lxx processors, this module may be added to the I/O tree while in Remote RUN. (And deleted on the run if no code is using its I/O buffers.)

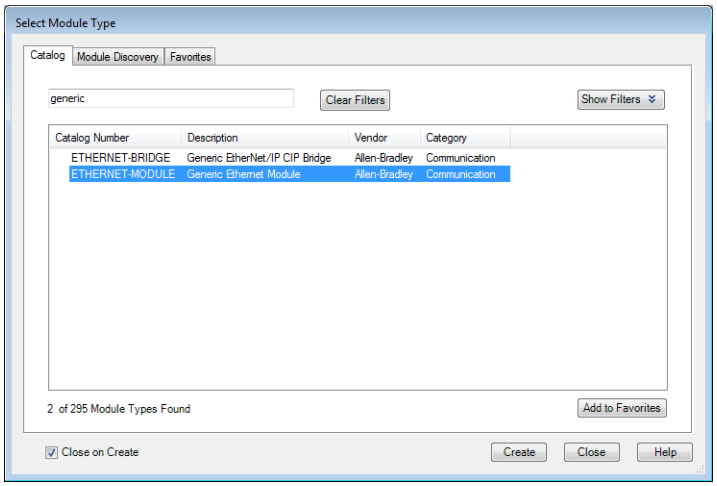


Figure 8: Generic Module Selection

In an RSLogic 5000 project, right click on the "Ethernet" icon in the I/O Tree and select "New Module..." to open the module search dialog. Enter "generic" in the search box to show the `ETHERNET-MODULE` and the `ETHERNET-BRIDGE` options. Select the ethernet module as shown in Figure 8, then click the "Create..." button to proceed.

Set the initial properties of the new module in the following dialog, as shown in Figure 9. If the scanning controller's logic will be directly accessing the module buffer as separate variables, be sure to select an appropriate Communication Format. The input and output data buffers will be set to that data type. For maximum flexibility when copying to and from structured data types, use one of the `SINT` communication formats.

Supply a module name, the IP address of the driver instance's Port, and the assembly numbers to use. Any of the buffer sizes may be set to zero to disable that function, but assembly numbers must always be provided. These values may be adjusted later from the "General" tab of the module's properties dialog, except for communication format.

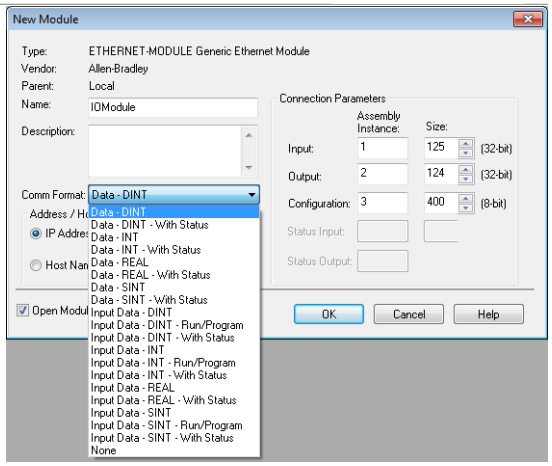


Figure 9: Logix Generic Ethernet Module

Note that the Configuration assembly buffer is only sent from the scanning controller when making the initial connection. It is limited to 400 bytes instead of 500 bytes, as the data for that buffer must fit in the same Forward Open request message that establishes the connection.

After configuring the general properties of the I/O connection, adjust the requested packet rate and other connection properties in the second tab of the module properties dialog, shown in Figure 10. If the version of Logix used allows, consider setting the "Unicast" checkbox to maximize the efficiency of the switches carrying your network traffic.

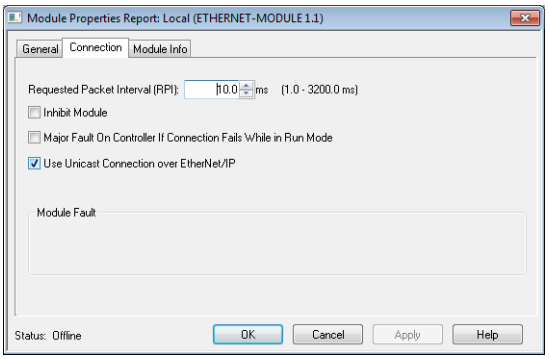


Figure 10: Logix Generic Connection Options

Select a packet rate that meets your application needs, but avoid going significantly faster, as Java's garbage collection algorithms can interfere with packet processing. Be aware that very high packet rates will consume substantial RAM and CPU time. Java's `MaxGCPauseMillis` configuration property should be set to a value less than or equal to the fastest RPI between the PLC and Ignition. Use java's detailed garbage collection logging facility to verify that your Ignition server can keep up with your packet rate. Inductive Automation's forums have guidance on optimizing and monitoring Java's performance, such as this forum thread.

For each assembly number you use, create the corresponding assembly in the driver instance, along with one or more Logix Tags to actually hold the data. The module creation dialog only offers arrays of basic data types for the scanning controller side of the connection, but you may use any array or structured data type on the Ignition side that suits your needs. In the scanning controller, the controller tags will have entries composed of the module name, a colon, then I, O, C, or S, depending on which assemblies you set up.

For the most predictable and maintainable data traffic, consider having the scanning controller contain a structured data type and controller tag for each I/O buffer. Use synchronous copy instructions (CPS) to move raw data from the generic module's input array to the structured input tag (at the beginning of the ladder code), and from the structured output tag to the output array (at the end of the ladder code). Use the same data types and tag names in the Ignition driver instance, pointing the Ignition side assemblies at the named structured tags. With this architecture, data written by ladder logic to the structured output tag will appear in Ignition with the same name, and data written by Ignition will arrive seamlessly in the ladder logic.

6.1.2 Multiple I/O Module Emulation with Logix Processors

When a single scanning controller needs more than a pair of 500-byte I/O buffers in your Ignition server, or there are other reasons to spread your connections across multiple driver instances, use backplane addressing in the virtual chassis to reach additional instances. In RSLogix, this functionality is provided by the `ETHERNET-BRIDGE` generic chassis – the other choice shown in Figure 8 above.

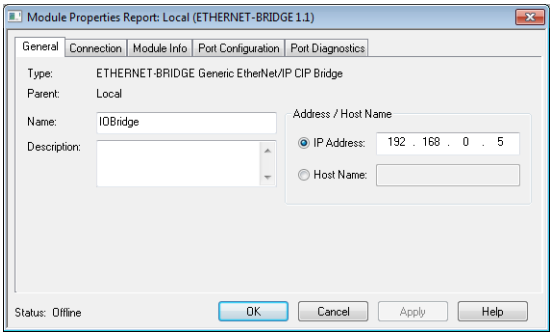


Figure 11: Logix Generic Chassis

The `ETHERNET-BRIDGE` module type exposes all driver instances, by slot number, to the scanning controller. When added to the I/O Tree, this module offers no data buffers of its own. Instead, it provides a named virtual backplane via a single IP address. It doesn't matter which driver instance has the IP address assigned to a port – all access will include slot number reference. After clicking "Create", complete configuration of the virtual chassis by supplying a name and IP address.

With the bridge module added to RSLogix, the individual driver instances can be added to the "CIP Bus". Right-click on the "CIP Bus", then select the `CIP-MODULE` (the only choice) to create. Provide details for the module as for the solo ethernet module, but provide a slot number instead of an IP address.

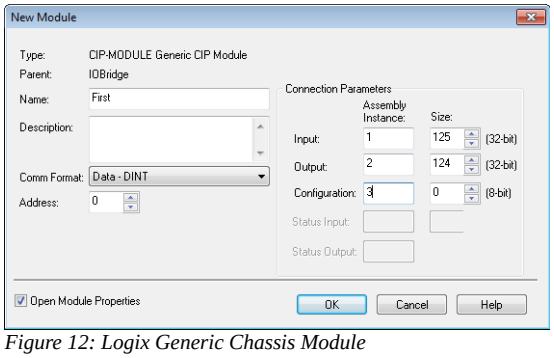


Figure 12: Logix Generic Chassis Module

Complete the module configuration via its "Connection" tab within the module properties dialog. Set an appropriate Requested Packet Interval and Unicast mode. Each virtual module on the backplane has its own communication setup.

Select a packet rate that meets your application needs, but avoid going significantly faster, as Java's garbage collection algorithms can interfere with packet processing. Be aware that very high packet rates will consume substantial RAM and CPU time. Java's `MaxGCPauseMillis` configuration property should be set to a value less than or equal to the fastest RPI between the PLC and Ignition. Use java's detailed garbage collection logging facility to verify that your Ignition server can keep up with your packet rate. Inductive Automation's forums have guidance on optimizing and monitoring Java's performance, such as this forum thread.

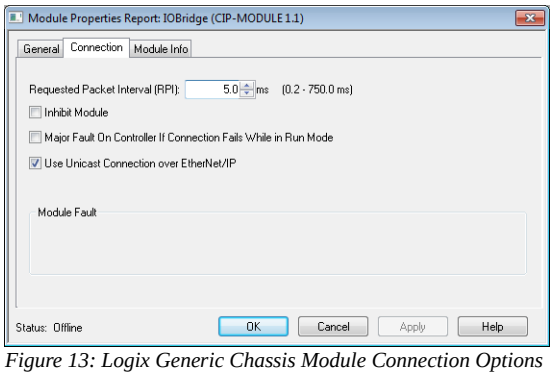


Figure 13: Logix Generic Chassis Module Connection Options

After configuration, each virtual I/O module's data buffers will be created in the processor's Controller Tags using the virtual chassis module name as a prefix, followed by a colon and the slot address, followed by :C, :I, :O, and/or :S, depending on the communication format and assembly sizes specified. Note that the module controller tags are listed under the virtual backplane's name and slot, not under the individual module's name. In the example shown, the module named "First" has controller tags: `IOBridge:0:C`, `IOBridge:0:I`, and `IOBridge:0:O`.

Remember that each virtual module in the virtual chassis must correspond to a driver instance in Ignition with that slot number and the driver instance must have assemblies set up the same as when using the

solo connection. If multiple identical modules are needed, use the Target Driver to populate multiple virtual slots with a shared XML configuration.

Consider a host driver XML configuration as shown in Figure 14, for an instance in virtual slot zero, where `MultiSlaveIn` and `MultiSlaveOut` are arrays of structures (one structure intended per slot).

```
<Controller>
  <DataTypes>
    <DataType Name="SlaveIO_t" Alignment="-4" Handle="e478">
      Bits=DINT
      Enable=HostedBy Bits.0
      Padding=DINT
      A=LINT
      B=DINT[2]
      C=INT[4]
      D=SINT[8]
    </DataType>
  </DataTypes>
  <Tags>
    <Tag Name="MultiSlaveIn" Definition="SlaveIO_t[100]"/>
    <Tag Name="MultiSlaveOut" Definition="SlaveIO_t[100]"/>
    <Tag Name="SlaveConfig" Definition="SINT[20]"/>
    <Tag Name="SlaveEcho" Definition="LINT"/>
  </Tags>
  <Assemblies>
    <Assembly Name="InputToMaster" CxPt="1001">
      <Member Bits="64" Target="SlaveEcho" />
      <Member Bits="320" Target="MultiSlaveIn[0]" />
    </Assembly>
    <Assembly Name="OutputFromMaster" CxPt="1002">
      <Member Bits="64" Target="SlaveEcho" />
      <Member Bits="320" Target="MultiSlaveOut[0]" />
    </Assembly>
    <Assembly Name="ConfigFromMaster" CxPt="1003">
      <Member Bits="160" Target="SlaveConfig" />
    </Assembly>
  </Assemblies>
</Controller>
```

Figure 14: Multiple I/O Module Host Example

{Tag data initialization bytes, Scanner module configuration, Jython code block, and programs blocks omitted for clarity.}

A Target Driver instance could then be configured as shown in Figure 15 to expose the rest of the slots (1-99) to the PLC, with payload redirection to/from the appropriate element of each array in slot zero.

```
<Controller>
  <Tags>
    <Tag Name="SlaveConfig" Definition="SINT[20]"/>
    <Tag Name="SlaveEcho" Definition="LINT"/>
  </Tags>
  <Assemblies>
    <Assembly Name="InputToMaster" CxPt="1001">
      <Member Bits="64" Target="SlaveEcho" />
      <Member Bits="320" Target="(slot 0)MultiSlaveIn[_slot]" />
    </Assembly>
    <Assembly Name="OutputFromMaster" CxPt="1002">
      <Member Bits="64" Target="SlaveEcho" />
      <Member Bits="320" Target="(slot 0)MultiSlaveOut[_slot]" />
    </Assembly>
    <Assembly Name="ConfigFromMaster" CxPt="1003">
      <Member Bits="160" Target="SlaveConfig" />
    </Assembly>
  </Assemblies>
</Controller>
```

Figure 15: Multiple I/O Module Target Example

Within the input and output assembly definitions, the payload members point into slot zero with a segment prefix and then use the predefined `_slot` variable as a dynamic subscript into the Host Driver configuration's array of structures.

With Rockwell PLCs' buffer size limits, and correspondingly larger structures than shown here, the total high speed traffic from PLC to Ignition could be as large as 49,600 bytes, while also sending 50,000 bytes back to the PLC. At short packet intervals.

6.1.3 Redundancy Considerations

When Ignition is operating as a redundant pair, both servers will be listening and responding on their respective IP addresses, but only the active gateway will supply live data. The PLC connections described above must be duplicated for the backup gateway, and there must be PLC logic that examines a heartbeat or other changing data from the gateway to determine which nodes to use.

6.2 Using Passive Messaging

In addition to, or instead of, I/O connections to Ignition, EtherNet/IP-equipped processors can use data table messaging with driver instances. When configuring a Logix message instruction, select CIP Generic and fill in the details for class, instance, attribute and service code, or use one of the pre-defined message types. CIP Data Table Read and Data Table Write are supported using the driver instance's tag names. When configuring the communications path (Figure 23), select a module you've placed in the I/O tree, or add and select one of the above virtual modules with Comm Format "None", or select the closest network component (ENBT, etc) then manually add to the connection path.

As with I/O connections, a redundant gateway pair will listen and respond at both the master and backup IP addresses. The tag reads from Ignition must be examined to determine which is the active gateway when both are running.

6.3 Using Producer Tags

All tags defined in the driver instance are automatically available as producer tags to any Logix or compatible processor that wishes to consume them. The data type in the Logix processor must be 500 bytes or less or the software will not permit configuring consumption at that end. The tag in the driver instance can be longer – only the beginning of the driver's tag will be transferred in that case.

Logix processors require that the target for their consuming tag show up in the I/O tree, with all intermediate nodes. If a Solo or Multiple Module I/O target is present in the I/O tree as described above, you may use the node in your Logix program for consumer tags as well. Otherwise, follow the instructions for either a Solo or Multiple Module setup but select "None" for the Comm Format.

On the Logix side, it is not possible to inhibit individual consumer tags. One can only inhibit nodes in the I/O tree, which inhibits all consumer tags pointed at or through that node.

6.3.1 Producing to Logix Structure Types

When a Logix PLC is connecting a structured consumer tag to your producer tag, it will make the connection request with a specific CRC in extra connection data. Ignition will ignore this CRC and deliver the tag data anyways. Ignition will also ignore any length mismatch, and will send exactly the number of data bytes requested, adding zero padding to the packet if needed. Ensuring that the structures on each end are byte-for-byte payload compatible is your responsibility.

6.3.2 Producing Data State Change Events

This driver has no way to suppress data state change events that occur when you (or anything else) writes into a producer tag. If you wish to not trigger fresh data events in a connected Logix event task,

do not allow any writes to that OPC item on the Ignition side. Structure your code to perform all such writes with a single `system.tag.writeBlocking()` or `system.opc.writeValues()` function call.

There is no way to restart the RPI timing for an Ignition producer tag. Select an RPI that minimizes the application latency when a change is transmitted.

6.4 Using Consumer Tags

When the Enhanced Host Driver is licensed (aka Scanner Option), Consumer Tags may be configured to receive data from Producer Tags in a Logix processor or compatible PLC. Each Consuming Tag connection is actually a form of I/O connection using User Datagrams, and is controlled in the driver instance by the Scanner Manager. The "Producer" setting in the tag configuration is the Route Path to the target Logix controller, the "Source Tag" setting is the Application Path in the Logix Controller (often just a tag name of a real Logix controller), and the RPI is the input RPI from the controller to the driver instance. The heartbeat output RPI is 2.5 times the input RPI.

When present, these settings cause the Scanner Manager to create additional Scanner instances with the settings needed to make the connection. These will show up in the OPC Browser under `Controller:Internals` => `Scanner Mgr`, by name, along with the named I/O Scanner modules.

The scanner instances created for consumer tags have a restricted list of attributes, as there is no configuration or output data to send, and the input data is automatically routed to the tag. Runtime connection monitoring and adjustment is otherwise the same. Note that consumer tags have a realtime mode Dword for special cases.

While connected, the latest packet consume timestamp is available at `TagName(attr 164)@usec`. These and other attributes (see Monitoring and Runtime Adjustment below) may be found in the OPC Browser under `Controller:Internals` => `Lgx Symbol Mgr`, by instance number and tag name.

6.4.1 Consuming Logix Structure Types

When the producer tag in the Logix processor is a structure type, particularly when carrying a `CONNECTION_STATUS` member at the beginning, the connection will fail for a simple Source Tag name. For such cases, use the following construct in the Source Tag setting:

```
(cls 105 inst 1)ProducerName(cxpt 1 data[1 4 0x02a0 0xWXYZ])
```

In this construct, replace `WXYZ` with the hexadecimal CRC of the data type. If you re-create the data type exactly in the Host Driver, it's CRC shown on the configuration page is reasonably likely to be correct. If not, you may need to temporarily point this module's Generic EtherNet/IP Driver at the target processor, and examine the probe log or the type's auto-generated JSON to obtain this value. (The hexadecimal will be simplified to decimal for you.)

Also use the above Source Tag construct when the tag is an array of structures..

6.4.2 Consuming Data State Change Events

Normally, class 1 traffic includes a 16-bit sequence number that the sender changes to indicate content has changed. With Logix output buffers and producer tags, by default, the number changes for any ordinary changes/copies/writes into the subject tags. However, Logix programs can have automatic output processing turned off, which ***suppresses*** the change notice for output buffers in the corresponding program. Similarly, a Logix Producer Tag can be configured to "Send Data State Change Events to Consumer(s)", with which it will ***not*** automatically send change notices to consumers. The PLC programmer must then use the `IOT` instruction to bump the sequence number manually on a per-tag or per-output-buffer basis.

On the Ignition side, there are two ways to see a state change event:

- Monitor the “Change Timestamp” available as `TagName(attr 163)@usec`
- Use a scripted consume event, and check the `.detail` field of the event. See Jython Data Events for further information.

Be aware that the Logix `IOT` instruction will cause an immediate packet transmission, and will restart the RPI timer.

6.5 Using Active I/O Scanning

Under an Enhanced Host Driver license (aka Scanner Option), direct scanning of EtherNet/IP I/O devices is available. This eliminates the need for a PLC in light-duty applications, and allows the use of listen-only connections to monitor EtherNet/IP devices that are slaved to an actual PLC.

The Module Scanner within a driver instance originates Input and/or Output connections using Transport Class 1, Scheduled priority, and Cyclic timing. Support for Change-of-State timing is planned for a future release.

Much of the information needed to complete this configuration is obtained from an I/O module’s EDS file. See the section on “Interpreting EDS Files” for details. The details will be entered in three panels of the New I/O Module and corresponding Edit page, for Identity, Connection, and Data settings.

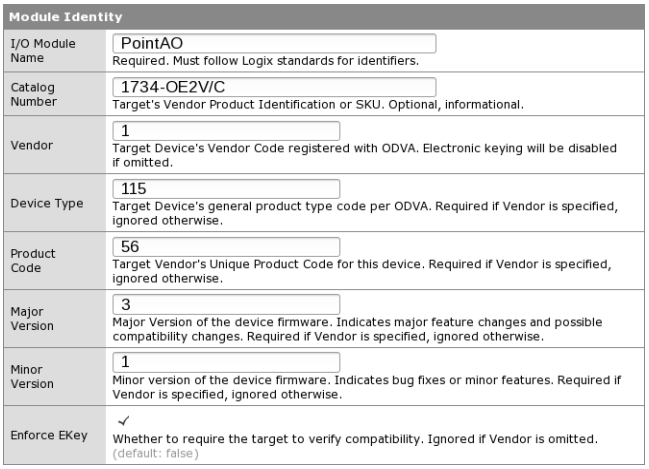


Figure 16: Scanner Electronic Keying

6.5.1 Target Module Electronic Keying

The first of three configuration blocks supplies all of the electronic keying information, if needed. The details for this section are automatically filled in if a real L5X is imported from RSLogix. Otherwise they can be obtained from the manufacturer’s Electronic Data Sheet file. Some non-compliant targets do not work if an electronic key is present, even if enforcement is turned off. For such cases, omit the vendor key to disable the feature entirely.

If the electronic keying information is present, the scanner module list on the main configuration page will show the expected EDS file name that Logix products use when registering a product in the hardware database. The files are usually stored in **C:\Program Files\Rockwell Software\RSCommon\EDS**, or **C:\Users\Public\Documents\Rockwell\EDS**, or a similarly-named folder. The expected file name displayed includes the minor number as the last two hexadecimal digits, but it is common for registered products to use “00” in the file name instead of the minor version. You can use this information to get the EDS file from an RSLogix install instead of hunting the web for it. (You are likely to need it.) The example in Figure 16 produces the file name **0001007300380301.eds**.

If the target module has an embedded EDS file available for upload, you can use the Client Driver to obtain it for you.

6.5.2 Target Module Connection Detail

The second of three configuration blocks supplies supplies the network path from the driver instance to the target device, and the Transport Class 1 protocol modes and speeds. Generally, the CIP Route starts with “port n ipaddr”, where “n” is the port number in the driver instance that is on the same subnet as the target device, or on the subnet of the first hop. And “ipaddr” is the IP address of that target/first hop. If that first hop is an `1756-ENxT`, `1734-AENT`, or similar chassis ethernet interface, the next tokens will be “slot m” to identify the target module. For the example in Figure 17, the first hop is a `1734-AENT` and the `1734-OE2V` is the second I/O module in the chassis.

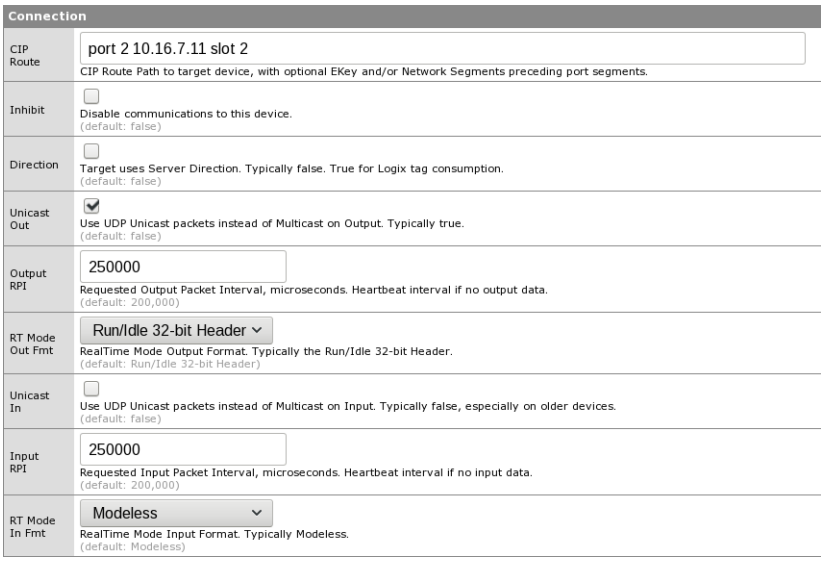


Figure 17: Scanner Route and Connection Details

The Inhibit checkbox prevents the driver instance from opening the connection. This is a writable attribute of the Scanner Module object under the Scanner Manager object, if you need runtime control of the connection. The Direction setting is used to manually configure the equivalent of a Consumer Tag, but to an alternate input object (an assembly, perhaps).

The Unicast, RPI, and RT Mode Format settings directly impact the Forward Open Request message, and must be compatible with the target object. Keep in mind that the RPI settings are in *micro*seconds, where RSLogix presents these as *milli*seconds with one decimal place. The driver instance will try to honor whatever is entered, though individual packets may be rounded off to the nearest millisecond, and Java garbage collection can interrupt.

6.5.3 Target Module Data

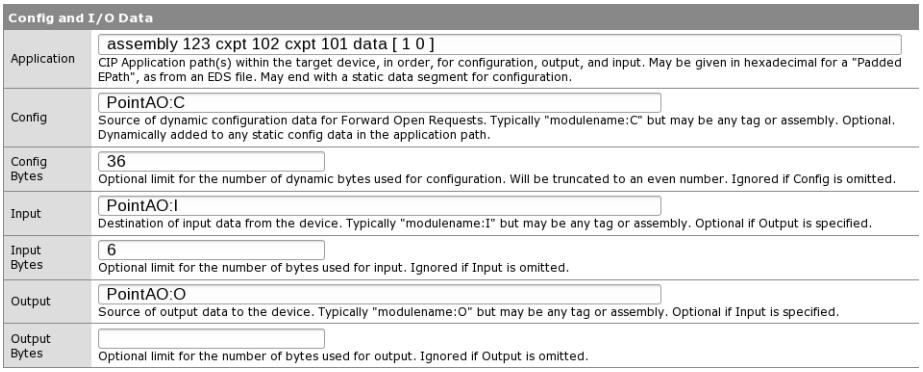


Figure 18: Scanner Data Configuration

Every I/O module to be scanned needs an Application Path that identifies the objects in the target device that will participate in the connection. The canonical format supplies four items: a config object path, an output object path, an input object path, and a configuration data segment, in that order. The CIP specification allows a device to offer any internal object as a config, output, or input item. The typical I/O device expects "assembly **c** cxpt **o** cxpt **i**", where **c**, **o**, and **i** are assembly numbers. See also the Application Path Segments topic.

The optional Config and Config Bytes settings control the construction of a configuration data segment to include in the Forward Open request message when the connection is started.

The Config setting is a tag path within the driver instance that contains the I/O module's configuration data. If an L5X file was imported, the tag name and data type will be appropriate, though the initial values may not be present. It is possible that the actual config data is shorter than the config tag, since Logix constructs all of its tags in increments of 32 bits. The Config Bytes settings limits the number of bytes inserted in the Forward Open request when the tag is larger than the module uses. (Consider using a structure with one of the two 16-bit alignment rules for Config Tag.)

Chassis-based I/O modules may expect a prefix to the config data that is consumed by the chassis. If constant and a multiple of 16 bits, a static data segment may be placed in the application path. This will be concatenated with the dynamic data from the Config tag. The example in Figure 18 shows the two 16-bit prefix words needed by the `1734-AENT` adapter to set up the `1734-OE2V` module.

At least one of the tags for Input and Output must be specified. If only an Input tag is given, the output packets will only be a heartbeat or contain the RealTime mode indicator. If only an Output tag is given, the input packets' content will be discarded.

Like the config tag, the size of the tag may be larger than the actual I/O module payload. The optional Input Bytes and Output Bytes settings allow the correct payload size to be specified. The example in Figure 18 shows an Input Bytes setting of six, required by the `1734-OE2V` module, overriding the tag size (eight bytes). If an L5X import was used, the tag settings can be expected to be correct, but the size limit settings might not be correct.

6.5.4 Monitoring and Runtime Adjustment

The bulk of the settings described above show up in the OPC Browser under `Controller:Internals => Scanner Mgr`, by name, and are live at runtime. Attributes 30 through 35 are read-only status values. The Entry Status, Fault Code, and Fault Information attributes correspond to the module information available in a Logix processor using the GSV instruction. The actual packet intervals are updated to reflect any adjustments the device made when it accepted the connection. An entry status of 0x4000, or 16384, signals a running connection.

Attribute #160, the RealTime Mode DWORD, is sent to I/O devices that use a Run/Idle header. It defaults to 0x1, indicating “Run Mode”. You may write a “0” to signal “Program Mode” to a connected output module. Each configured I/O scanner maintains an independent “Run Mode” value.

Writing to attributes 1 through 16 changes the parameters of the connection, and generally forces the connection to close and re-open.

6.5.5 Redundancy Behavior

Scanners defined above will be held in a disabled state on the standby gateway of a redundant pair.

6.6 Using the I/O Momentary Button

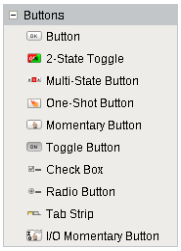


Figure 19: Designer Palette

This component is added to the designer’s palette of Vision Components at the end of the Buttons section, as shown in Figure 19. After placing one on your window in the designer, you must set the Virtual Device Name and Target Tag Path properties to identify the boolean to control. This is *not* a SQLTag address, but the device name and address within the EtherNet/IP module’s virtual backplane. This pushbutton *bypasses* the SQLTags subsystem. These properties are not case-sensitive.

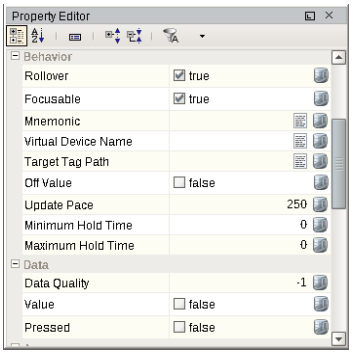


Figure 20: I/O Button Properties

Set the Off Value property to True to emulate a Normally-Closed pushbutton (e.g. typical Stop buttons). Set the update pace to match, or close to match, the RPI that will be used in the Gateway-to-Controller communications. Then check the rest of the timing properties’ defaults and adjust to suit your application. A minimum hold-time of five times the RPI is recommended for Stop buttons (Logix controllers will fault an I/O connection after four missed packets).

When a user is actually pressing the button at runtime (or in preview mode), background logic will repeatedly check that the mouse is still pressing the button (or the spacebar is still down if using the keyboard). If all is still well, the background function will ask the UI thread to send a ping to the gateway to report that the button is still pressed. This ensures that a frozen UI also kills the ping message. The gateway will monitor and enforce the hold time requirements, treating overlapping button presses from multiple clients as a single timed event. The gateway will expire a button press from a specific client if that client fails to ping within twice (x2) the Update Pace. The gateway will also report actual button value changes to all registered buttons (for a given address), whether pressed or not.

7 Host and Target Drivers' Functional Description

The Host Driver yields a single complete EtherNet/IP device in a given slot in the virtual backplane, with the required CIP objects and several optional objects, plus the objects emulating a Logix processor's Data Access objects. The Host Driver includes

7.1 Object Classes, Instances, and Attributes

The CIP specification requires compliant devices to implement a minimum set of object-oriented services, with a set of specified object classes with well-defined attributes and behaviors, plus vendor-defined objects and attributes. Objects are nested in a hierarchy of classes, class attributes, instances, and instance attributes, with further nesting of complex data types. As shown in Figure 21 below, the Message Router is the foundation of the driver instance, and is directly exposed to Ignition's OPC server.

The message router provides access to all other object classes and their instances and attributes using path segments, described in detail below, to navigate the hierarchy and select a specific object or fragment thereof. The interface to Ignition's OPC server translates node address strings into the internal path segments needed by the message router, and supplies appropriate node address strings to the user when browsing the hierarchy.

CIP Specification version 3.19 distinguishes between "device scope" objects and "port scope" objects, with the Connection Manager and Connection List classes assigned to the per-port scopes. Figure 21 shows the unique connection manager and connection list per communications port. The unconnected message manager (UCMM) in each port is the entry point from the outside. Later versions of the CIP Specification relax these port isolation rules, but this module implements them.

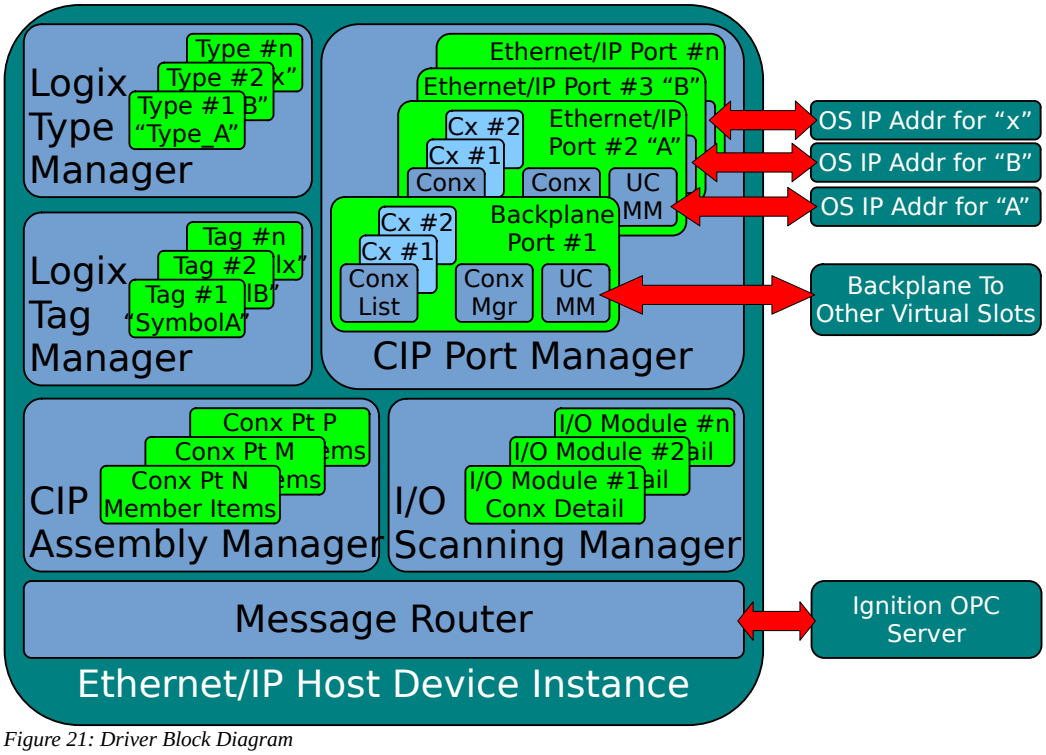


Figure 21: Driver Block Diagram

The Target Driver is similar to the above, except its Port Manager only has the backplane, port #1, connecting to other virtual modules. A single Target Driver instance fills multiple slots with identical virtual modules, all connected just to the backplane.

7.2 EtherNet/IP Encapsulation in TCP/IP

All communication in the EtherNet/IP specification starts with a TCP/IP connection from an originator to a target's UCMM in its ethernet port. The unconnected message manager can execute CIP requests in both port and device scope. The total size of a request and its response must each fit in 504 bytes. If an ad-hoc request needs to be directed to another port, or out another port to a bridged target, the originator can wrap the request in an Unconnected Send message.

When a Logix processor executes a message instruction where the "Connected" checkbox is turned ***off***, and there's any route path beyond the first IP address, the message will be wrapped in an unconnected send with the balance of the route path. Each hop will pop off one route path element, keeping the request wrapped, until the final UCMM can handle the message directly. Each hop will keep track of the outstanding requests it has passed on so the responses can go back to the correct originator. A message sent this way must be small enough that the unconnected send header, the wrapped message, and the intermediate route path will all fit within the 504 byte limit.

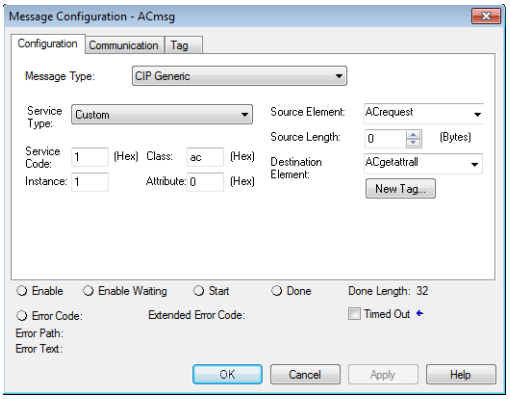


Figure 22: Logix Message Configuration

When a Logix processor executes a message instruction where the "Connected" checkbox is turned ***on***, it briefly defers the message until it can construct a routed CIP connection to the final endpoint's Message Router using "Transport Class 3". This operation starts with a Forward Open request to the first hop's connection manager containing the connection parameters and final target's route path. When the first hop reports this has succeeded, each hop along the route has established buffers and assigned shortcut "connection IDs" to a connection. Which then allows full-size message requests and full-size responses to flow back and forth. The Logix processor can then send the original request message down this connection, inside the TCP/IP channel.

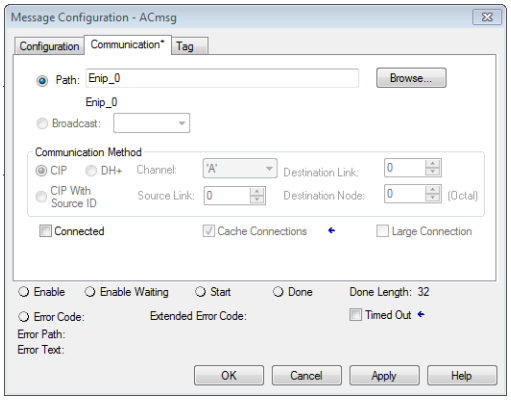


Figure 23: Logix Message Connection

A Logix processor offers two optimizations when connected messaging is used: 1) caching, and 2) large buffers. The "Cache Connections" checkbox determines if the Logix processor will delete the connection soon after the message completes, or if it will hang on to the connection for several seconds in the hope that other message instructions will need to talk to the same endpoint. Or the same message is repeated soon. This dramatically improves message response times. The "Large Connection" checkbox instructs the processor to use ~4k message buffers within the TCP/IP channel instead of the normal 512-byte buffers. This is an optional feature in the specification that neither very old processors nor many I/O devices support.

7.3 EtherNet/IP Encapsulation in UDP/IP

I/O traffic in EtherNet/IP is carried in User Datagrams, aka UDP/IP, not inside a TCP/IP connection. User Datagrams also carry Producer/Consumer tag data. User Datagrams allow processors and I/O modules to ignore the occasional lost packet and keep going, as long as the gap isn't too large. When a TCP/IP channel loses a packet, it is obligated to report the missing packet and get it retransmitted before continuing with any other data delivery. (TCP/IP ***itself*** requires this—it isn't a requirement added by

EtherNet/IP or other protocols.) This is the root cause of many hiccups in data delivery for most HMIs and typical SCADA communications.

When a Logix processor or other EtherNet/IP scanner establishes an I/O connection, it starts by opening a TCP/IP connection to the first hop in the route and then issuing a Forward Open request message to that device's connection manager. The connection parameters in this case specify "Transport Class 1", the target endpoints, the timing requirements, any custom IP socket details, and an optional payload for configuration. After the success message is returned inside the TCP/IP channel, UDP data traffic commences.

The TCP channel can expire while the UDP traffic runs, if no other connection needs the first hop UCMM. A follow-up Forward Open request may be generated (also via TCP/IP) if the data for the configuration assembly changes. UDP data traffic continues until either 1) a Forward Close request is sent via TCP/IP, or 2) either direction times out on consecutive lost packets. Missing UDP packets do not generate any kind of retransmit request. The receiver simply waits for the next scheduled packet. When Logix processors connect to network I/O, they generally specify four (4) missed packets as the broken connection limit.

8 Element Path Segments

Element Paths in the CIP specification are sequences of binary encoded keys that select an object or data item within an EtherNet/IP device (or other CIP devices, like DeviceNet or ControlNet). The specification refers to the individual encoded keys as Path Segments. The CIP specification has segment types for a variety of items, including the class/instance/attribute values in the object model described above, as well as symbols, array subscripts, and bit numbers within integer data. There are also segment types that declare or define data types.

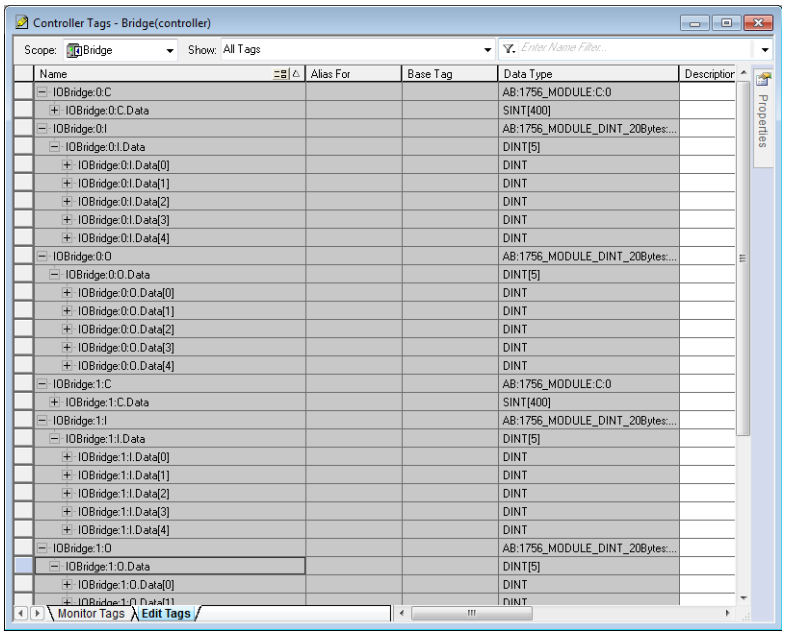


Figure 24: Ethernet Bridge Controller Tags

Since this module's applications are dominated by the tags and types used within Allen-Bradley's Logix family and Omron's NJ/NX family, this module's simplest address formats mimic the syntax a programmer would use for a controller tag in ladder logic. OPC Item paths for this module are the string formats for application paths as described below.

8.1 Path Segment Strings and Encodings

The CIP Specification, Volume 1 Appendix C, defines how path segments are encoded "on the wire". The first byte of encoded segments is a segment type code. This module can encode and decode nearly all of the segment types defined by the specfication, and can both display human-friendly string representations and parse those string representations.

This module also repurposes several of the reserved type codes as private extensions (along with some reserved secondary codes for defined constructed types) in order to handle some of the Omron datatypes (that otherwise clash with CIP port segments), to support enhanced formal structure definitions, and to distinguish between requirements for padded epaths versus their packed form.

8.1.1 Element Path String formats

While defined in the specification individually, most uses of path segments involve multiple selectors, possibly including data definition segments. Since both humans and Ignition's OPC server present address information as strings, there needs to be a reliable conversion from string to list of path segments, and the inverse conversion for path segments decoded from the wire or otherwise obtained internally.

As noted above, the format of path strings that will be used in OPC Item Paths and elsewhere throughout the module is intended to closely conform to the syntax a PLC programmer would expect within their programming environment. Paths starting with a symbol, and composed of symbols and subscripts and bit numbers, with dots and square bracket punctuation, are the common form for both Allen-Bradley and Omron, and are directly parsed in this module. This simple form, with no whitespace nor extraneous punctuation, having only unsigned decimal subscripts or bit numbers, is called a "tagpath", and any collection of path segments that can be reversibly represented this way will be so rendered. Such paths are composed entirely of (ansi) symbol segments, member segments, and bit index segments.

However, there are many more segment types, including the class, instance, and attribute segments that dominate generic CIP messaging. To handle the rest of the segment types, the string parser and string renderer work with “token strings”, where specific keywords identify segment types, and arguments supply any other information a segment type uses. When used together with tagpaths, or when ambiguous, the token string form is surrounded by parentheses.

Tokens are separated by whitespace. In a token string, if any token ***argument*** needs whitespace or punctuation or special characters, it can be enclosed in single or double quotation marks. Quoted token arguments may use standard backslash escapes for common special characters.

A tagpath starts with a symbol, and is optionally followed by decimal subscripts enclosed in square brackets and more dot-delimited symbols, in any order, and optionally ending with a dot-delimited decimal bit number. Multiple consecutive subscripts may be comma-delimited within one set of square brackets. No whitespace is permitted. Some examples:

tag

tag[3,7]

tag.member

tag.member[25]

tag.member[3].submember

tag.member[0].5

The complete segment path string syntax combines one or more tagpaths and token strings, like the following patterns:

tagpath

tokens and args[1]

(tokens)

(tokens)tagpath

tagpath(tokens)

tagpath[2](tokens)tagpath...

(tokens)tagpath(more tokens)tagpath...

When displaying segment paths, in most cases, the renderer will produce the shortest available form.

8.1.2 Segment Type Tokens Index

With tokens for secondary forms, and abbreviations and macros, the following are all of the keywords recognized by the string parser:

aliasof align1 align2 align2c align4 align4l align8 align8l ansi
assembly assy attr attribute bcd2 bcd4 bcd8 bit bool bool16
bool32 bool64 byte class cls connection conx conxpoint cxmgr cxpt
data date date_and_time defstruct dint dt dword ekey engunit
epath eth_mac_addr ethmac fixedtag ftag ftime hostedby identity
inh inhibit inst instance int itime lint lreal ltime lword memb
member memberhandle membernumber membhand membnum msgrouter ntime
null odo odometer omdatens omdt omron_date_and_time
omron_time_of_day omrondt omrontod omtimens omtod padbyte

1 To omit the parentheses, the token string must have white space or an equal sign before any other punctuation.

2 When token strings follow a tagpath, that preceding tagpath cannot end with a bit number.

padepath param parameter port real sched schedule service sgnodo short_string shortstring signed_odometer sint slot stime string string2 stringi struct sub subscript svc symbol tag template time time_of_day tod tpl udint uint ulint usint utime word

Except as noted below for Automation Professionals' private extensions, and the extensions to support Omron's vendor-specific types, encodings are per the CIP specification.

8.1.3 Segment Type Code Summary

Type codes (first byte of the encoded form) are summarized here. Shortest token abbreviations shown. Italics are for implicit forms. Gray shading is for reserved values, gold shading for Automation Professionals' extensions, and rose shading for unimplemented values:

Code: Token	Code: Token	Code: Token	Code: Token
0x00: port 0 n	0x01: slot n	0x02: port 2 n	0x03: port 3 n
0x04 through 0x0b: single byte addresses for ports 4 through 11			
0x0c: port 12 n	0x0d: port 13 n	0x0e: port 14 n	0x0f: port p n
0x10: port 0 aa	0x11: slot aa	0x12: port 2 aa	0x13: port 3 aa
0x14 through 0x1b: multi-byte addresses for ports 4 through 11			
0x1c: port 12 aa	0x1d: port 13 aa	0x1e: port 14 aa	0x1f: port p aa
0x20: cls c {8bit}	0x21: cls c {16bit}	0x22	0x23
0x24: inst i {8bit}	0x25: inst i {16bit}	0x26: inst i {32bit}	0x27
0x28: memb m {8bit}	0x29: memb m {16bit}	0x2a: memb m {32bit}	0x2b
0x2c: cxpt p {8bit}	0x2d: cxpt p {16bit}	0x2e: cxpt p {32bit}	0x2f
0x30: attr a {8bit}	0x31: attr a {16bit}	0x32	0x33
0x34: ekey k...	0x35	0x36	0x37
0x38: svc s {8bit}	0x39	0x3a	0x3b
0x3c: *ext* e n {8bit}	0x3d: *ext* e n {16bit}	0x3e: *ext* e n {32bit}	0x3f
0x40:	0x41: sched d {8bit}	0x42: ftag d {8bit}	0x43: inh ms {8bit}
0x44 through 0x4f			
0x50: safety s {n byte}	0x51: inh µs {n byte}	0x52: auth a {n byte}	0x53
0x54 through 0x5e			0x5f: *ext net* {n byte}
0x60: symbol s {n byte}	0x61 through 0x7f: symbol s {1-31 byte}		
0x80: data d {n byte}	0x81 through 0x8f		
0x90	0x91: ansi s {n byte}	0x92 through 0x9f	
0xa0: struct n	0xa1: *abbrev array*	0xa2: *formal struct*	0xa3: *formal array*
0xa4: align1 *struct*	0xa5: align2 *struct*	0xa6: align4 *struct*	0xa7: align8 *struct*
0xa8: handle struct	0xa9: align2c *struct*	0xaa: align4l *struct*	0xab: align8l *struct*
0xac: *dynamic array*	0xad: hbit h.b	0xae: struct s	0xaf: aliasof *tagpath*
0xb0: defstruct c	0xb1	0xb2	0xb3
0xb4: bcd2	0xb5: bcd4	0xb6: bcd8	0xb7: enum
0xb8: omdatens	0xb9: omtimens	0xba: omdt	0xbb: omtod
0xbc: union	0xbd	0xbe	0xbf: omstring
0xc0: utime	0xc1: bool	0xc2: sint	0xc3: int
0xc4: dint	0xc5: lint	0xc6: usint	0xc7: uint
0xc8: udint	0xc9: ulint	0xca: real	0xcb: lreal
0xcc: stime	0xcd: date	0xce: tod	0xcf: dt
0xd0: string	0xd1: byte	0xd2: word	0xd3: dword
0xd4: lword	0xd5: string2	0xd6: ftime	0xd7: ltime
0xd8: itime	0xd9: stringn	0xda: shortstring	0xdb: time
0xdc: epath	0xdd: engunit	0xde: stringi	0xdf: ntime
0xe0 through 0xe3: future elementary types			
0xe4 through 0xff			

Type code 0x34 for Electronic Keying has two specified secondary key type codes, 4 and 5. Only key type 4 is supported.

Type codes 0x3c through 0x3e for Extended Logical Segments have six specified secondary type codes. All are supported.

Type code 0xb0 for Defined Constructed types has three specified codes for subtypes, all supported, plus four private extension subtypes.

8.2 Route Path Segments

Path segments in this group are used throughout the CIP object model to indicate the path from device to device and to control communications between devices and their application objects.

Port segments are used within Route paths to define the hops from one device to another. Route paths can also include Electronic Key segments and Network Segments.

8.2.1 Port Segments

Syntax is “port n a”, where $0 \le n \le 65535$ and either $0 \le a \le 255$, or “a” is an ASCII character string of of zero to 255 bytes. Port numbers less than 15 are encoded with an abbreviated form, as are numerical addresses between 0 and 255. Alternate syntax “slot a” indicates port #1 and requires $0 \le a \le 255$. The convention is to supply IP addresses or host names in ASCII form.

8.2.2 Electronic Key Segments

Type code is always 0x34. The following byte is a key format code. Only format #4 is supported at this time. The payload for format #4 is eight bytes, containing vendor, device type, product code, and revision. Syntax is “ekey v d p mj.mn c”, where “v”, “d”, and “p” are unsigned 16-bit values, “mj” is an unsigned 7-bit value, “mn” is an unsigned 8-bit value, and “c” is one of “enforce”, “allow”, “0”, or “1”. See CIP Volume 1 Table C-1.5 in §C-1.4.2 for details.

8.2.3 Network Parameter Segments

The specification provides several segments to be used in route paths to alter network behavior in special cases. Note that the token string form of the inhibit segment always uses integer microseconds. The millisecond format with type code 0x43 is automatically selected if the microsecond value is compatible. Details are in CIP Volume 1 §C-1.4.3.

8.3 Application Path Segments

Path segments in this group are used throughout the CIP object model to indicate the path within a device to desired objects or data fragments within objects. In some contexts, they can carry configuration information that will apply to later data transfer.

Character String segments, and Logical segments indicating Class, Instance, Attribute, and Member, are the most common selectors within Application Paths, as the target path of a CIP Request within a device.

Some of these segments are not commonly used "on the wire", but instead direct fine-grained selection within a larger data block that **is** actually transferred.

8.3.1 Numeric Logical and Extended Logical Segments

Multiple keywords with similar syntax. Syntax is "*keyword* n" and "n" is an unsigned integer that fits within the segment type's max size. Encoding is always normalized to the smallest format needed for the value. Decoding tolerates non-normalized values. Details are in CIP Volume 1 §C-1.4.2. Keywords are:

Type Codes	Secondary Type	Keyword	Abbreviated KW	Size
0x20, 0x21		class	cls	16-bit
0x24, 0x25, 0x26		instance	inst	32-bit
0x28, 0x29, 0x2a		member	memb	32-bit
0x2c, 0x2d, 0x2e		conxpoint	cxpt	32-bit
0x30, 0x31, 0x32		attribute	attr	16-bit
0x38		service	svc	8-bit
0x3c, 0x3d, 0x3e	0x01	subscript	sub	32-bit
0x3c, 0x3d, 0x3e	0x03	bit		32-bit
0x3c, 0x3d, 0x3e	0x05	membernumber	membnum	32-bit
0x3c, 0x3d, 0x3e	0x06	memberhandle	membhand	32-bit

Several keywords are provided as macros to produce the most common combinations of class and instance logical segments, as follows:

Macro	Alternate Macro	Expands To
identity		class 1 instance 1
msgrouter		class 2 instance 1
cxmgr		class 6 instance 1
assembly N	assy N	class 4 instance N
connection N	conx N	class 5 instance N
parameter N	param N	class 15 instance N
tag N	*tagpath*	class 0x6b instance N
template N	tpl N	class 0x6c instance N

When converting a complete application path to string form, these macros will be inserted where applicable to achieve the shortest form.

8.3.2 Indirect Extended Logical Segments

Keywords "subscript" and "bit" shown above can also parse a tagpath or token string argument instead of a numeric value. In this case, type code 0x3e is not allowed, the secondary types will be 0x02 and 0x04, respectively, and the encoded number is the number of words to follow containing the padded EPath. The nested path is always in padded form, even if embedded within a packed EPath. As above, details are in CIP Volume 1 §C-1.4.2. If the nested path is not a simple string, enclose it in quotation marks.

8.3.3 Symbol and Data Selection Segments

Character selectors and data selectors carry a variable-length payload with the selection information.

Type Codes	Syntax	Functionality
0x60-0x7f	symbol s	Character string selector with alternate charsets, length 1-31 characters (not bytes).
0x80	data d	Embedded data selector with single 16-bit value.
0x80	data [d e...]	Embedded data selector with up to 255 16-bit values.
0x91	ansi s	Character string selector with ISO-8859-1 charset, max length 255 bytes. (Often used implicitly with UTF-8.)

The preferred format for character string selectors in most applications is type code 0x91. (Many applications do not support type codes 0x60-0x7f at all.)

The embedded data selector is most commonly used to supply configuration data for an I/O connection within a Forward Open request. In that usage, the length is limited to 200 16-bit values.

8.3.4 Special Symbol Segments

This module uses a number of special symbols via the OPC interface. They all begin with the "@" character. For most, these are included as a trailing symbol in a tagpath (delimited with a period if necessary, as usual), or using the ansi token, but will not be passed on the wire to a target device—they are stripped off and interpreted by the driver. The following special symbols are supported:

Syntax	Functionality
@barrier	Solo tagpath symbol that yields a timestamp when the entire batch of reads that it accompanies completes. Not writable.
@debug @trace	These two symbols may be placed anywhere in an application path. They are stripped out before optimization, but their presence is noted, and causes corresponding loggers to switch to INFO level wherever the driver handles the item.
@cache	Prefix that indicates the following element path should be read entirely from the OPC layer's cache. Applies only to probed class/instance/attribute values acquired during startup probing. Not writable.
@string	When applied to an array of 8-bit values, reads the contents as a null-terminated string or writes a null-padded string. Uses the CIP default charset, ISO-8859-1.
@utf8	When applied to an array of 8-bit values, reads the contents as a null-terminated string or writes a null-padded string. Uses the UTF-8 charset.
@utf16	When applied to an array of 16-bit values, reads the contents as a null-terminated string or writes a null-padded string. Uses the UTF-16 charset.
@msec	When applied to a 64-bit integer type, reads the value as UTC microseconds and returns an OPC DateTime, or writes UTC microseconds from an OPC DateTime.
@nsec	When applied to a 64-bit integer type, interprets the value as UTC nanoseconds and returns an OPC DateTime, or writes UTC nanoseconds from an OPC DateTime.

8.3.5 Keyence Application Paths

When a target device is a Keyence KV PLC, addresses within its global memory use more special symbols, combining the device memory mnemonic with the numeric or hex address. They all begin with the "@" character. (Internally, these are split into a mnemonic symbol segment plus a binary member segment.) The follow Keyence memory device application paths are supported:

Syntax	Functionality
@Rnbb @MRnbb @LRnbb @CRnbb	Boolean relays, auxiliarly relays, latch relays, and control relays, respectively. For each, 'n' is the zero-based decimal channel number, and 'bb' is the decimal bit (00-15) within the channel.
@Bx @VBx	Boolean link relays and work relays, respectively. In each pattern, 'x' is the zero-based hexadecimal linear bit address.
@DMn or @Dn, @DMn(t) or @Dn(t) @EMn or @EMn(t) @FMn or @FMn(t) @ZFn or @ZFn(t) @TMn or @TMn(t) @Cmn or @CMn(t) @Vmn or @VMn(t)	Data memory, extended data memory, paged file memory, unpaged file memory, temporary data memory, control memory, and work memory, respectively. For each, 'n' is the zero-based decimal linear address, and 't' is an optional type override. When 't' is absent, 16-bit INT is used. 't' may be UINT, DINT, UDINT, REAL, LINT, ULINT, or LREAL. Consecutive addresses, in little-endian order, are used with the larger data types. The parentheses are required if 't' is present.
@Wx or @Wx(t)	Link memory. 'x' is the zero-based hexadecimal linear address. 't' is an optional type override, as described for data memories.
@Tn.DN, @Tn.ACC, or @Tn.PRE @Cn.DN, @Cn.ACC, or @Cn.PRE @CTHn.DN, @CTHn.ACC, or @CTHn.PRE @CTCn.DN, @CTCn.ACC, or @CTCn.PREZn	Timers, Counters, High-Speed Counters, and High-Speed Counter Comparators, respectively. These must always be specified with one of the given suffixes (Done, Accumulator, or Preset). The CC, CS, TC, and TS mnemonics for simplified access to accumulators and presets are not supported.
@Zn @ATn	Index registers and digital trimmers, respectively. These are always unsigned 32-bit values.

8.4 Data Definition Segments

Path segments in this group do not perform object or data element selection, but define the format of data blocks that appear in CIP request or reply payloads, or the data areas of implicit messaging packets. In essence, they provide the foundation of arbitrary encoders and decoders that can be constructed at runtime, instead of hard-coded into base software.

When the Client Driver probes a PLC, it translates the PLC probe results into byte-for-byte equivalent codecs using the tools in this group. The Client Driver also reads XML that declares the data types for many of the attributes within standard object models from the CIP Specification. This permits proper encoding/decoding of those objects' attributes.

Similarly, when the Host Driver loads XML to define its emulated types and tags and assemblies, it sets up the tools in this group to control the encoding and decoding of packet payloads.

8.4.1 Elementary Data Segments

Elementary data definition segments are encoded with a single byte, and represent a single, indivisible unit of data. The data represented by these segments is transferred in little-endian format unless otherwise noted. Elementary types include the basic numeric types and some complex items that are expected to be treated as single values. The token keywords take no arguments.

Omron data type details are per Software User's Manual W501 §6-3-5 and per EtherNet/IP Port User's Manual W506 §8-7-1. CIP data type details are per Volume 1 §C-2.1.1 and §C-6.1. Note that CIP Volume 1 §C-2.1.2 specifies ISO 8859-1 for strings built from single bytes (except where the character set is explicit in the payload of a StringI value), but this module defaults to UTF-8 instead to conform to typical usage in real-world applications.

Type Code	Keyword	Abbreviated	Data Size	Data Format and functionality
0xb4	bcd2		16-bit	Omron BCD. Code 0x04 on the wire.
0xb5	bcd4		32-bit	Omron BCD. Code 0x05 on the wire.
0xb6	bcd8		64-bit	Omron BCD. Code 0x06 on the wire.
0xb8	omdatens		64-bit	Omron epoch nanoseconds, but time of day ignored. Code 0x08 on the wire.
0xb9	omtimens		64-bit	Omron signed nanoseconds duration. Code 0x09 on the wire.
0xba	omron_date_and_time	omdt, omrondt	64-bit	Omron epoch nanoseconds. Code 0x0a on the wire.
0xbb	omron_time_of_day	omtod, omrontod	64-bit	Omron nanoseconds since midnight. Code 0x0b on the wire.
0xc0	utime		64-bit	CIP epoch microseconds.
0xc1	bool		8-bit	CIP boolean, 0 or 1 only. Arrays of bool pack into one or more bytes.
0xc2	sint		8-bit	CIP signed short integer.
0xc3	int		16-bit	CIP signed integer.
0xc4	dint		32-bit	CIP signed double integer.
0xc5	lint		64-bit	CIP signed long integer.
0xc6	usint		8-bit	CIP unsigned short integer.
0xc7	uint		16-bit	CIP unsigned integer.
0xc8	udint		32-bit	CIP unsigned double integer.
0xc9	ulint		64-bit	CIP unsigned long integer.
0xca	real		32-bit	CIP floating point, IEEE 754.
0xcb	lreal		64-bit	CIP floating point, IEEE 754.
0xcc	stime		64-bit	CIP epoch nanoseconds.
0xcd	date		16-bit	CIP unsigned days since 1972-01-01.
0xce	time_of_day	tod	32-bit	CIP milliseconds since midnight.
0xcf	date_and_time	dt	48-bit	CIP milliseconds since 1972-01-01 (concatenated tod and date elements).
0xd0	string		variable	CIP string of bytes, with a 16-bit length prefix.
0xd1	byte		8-bit	CIP bit string.
0xd2	word		16-bit	CIP bit string.
0xd3	dword		32-bit	CIP bit string.
0xd4	lword		64-bit	CIP bit string.
0xd5	string2		varies	CIP string of words, with a 16-bit length prefix.
0xd6	ftime		32-bit	CIP microseconds signed duration.
0xd7	ltime		64-bit	CIP microseconds signed duration.
0xd8	itime		16-bit	CIP milliseconds signed duration.
0xda	short_string	shortstring	variable	CIP string of bytes, with an 8-bit length prefix.
0xdb	time		32-bit	CIP milliseconds signed duration.
0xdc	epath		variable	CIP encoded path. Packed form. Converted to and from string format on the OPC side.
0xdd	engunit		16-bit	CIP engineering unit per CIP Volume 1 Appendix D.
0xde	stringi		variable	CIP internationalized string, multiple languages.
0xdf	ntime		64-bit	CIP nanoseconds signed duration.

8.4.2 Predefined Structure Segments

The CIP specification's "Defined Constructed Data Types" use type code 0xb0, followed by a 16-bit integer subtype. Details are per CIP Volume 1 §C-2.1.5 and §C-6.2.3. Note that the odometer types are not particularly useful when 64-bit integers are available, as a 64-bit integer has a greater range of values. This module converts to/from 64-bit integers on the OPC side.

This module repurposes four reserved subtypes as private extensions in order to implement Omron's boolean type, similar boolean types extrapolated to larger underlying storage, and to make it possible to dynamically define decoders for all of the service payloads for all object attributes in the CIP object model. This requires distinguishing between packed and padded EPaths, and enforcing word alignment for some services.

SubType	Keyword	Abbreviated	Data Size	
0x01	signed_odometer	sgnodo	10 bytes	CIP extra-long signed integer, composed of five 16-bit signed integers, each constrained to -999≤x≤999.
0x02	odometer	odo	10 bytes	CIP extra-long unsigned integer, composed of five 16-bit integers, each constrained to 0≤x≤999.
0x03	eth_mac_addr	ethmac	6 bytes	CIP ethernet MAC address storage, in network order.
0xc7	bool16		16-bit	Omron native boolean. Functionally identical to bool, but uses WORD instead of BYTE for storage and transfer.
0xc8	bool32		32-bit	As above, but with DWORD storage and transfer.
0xc9	bool64		64-bit	As above, but with LWORD storage and transfer.
0xdc	padepath		varies	CIP encoded path. Padded form. Converted to and from string format on the OPC side.

8.4.3 Array Prefix Segments

The CIP Specification defines an abbreviated format for array definitions. This is unimplemented in this module, as it appears to have no real-world use. This module implements formal array definition segments as a prefix to other data definition segments, up to three levels.

Type code 0xa3 is implemented per the specification. However, as a private extension it accepts an optional following member segment before the actual nested data definition segment. When present, this member segment declares the starting subscript value, to support non-zero-based arrays.

Type code 0xac, reserved in the specification, is repurposed in this module as a private extension to encode an indirect array length, similar to the indirect subscript and indirect bit index selector segments. Where type code 0xa3 is immediately followed by a constant 32-bit dimension length, type code 0xac is followed by a byte indicating the number of bytes following in a packed EPath. It is then followed by a data definition segment like type code 0xa3. It does not support non-zero initial subscripts.

This implementation does not accept type code 0xac in any position other than the outermost dimension of a definition.

While the encoded form of these segments use prefixes, the token string forms use square bracket suffixes, similar to the appearance of tag path array subscripts. Which array type code to use is implied by the contents of the square brackets, as follows:

Type Code(s)	Syntax	Functionality
0xa3,...	type[n]	1D array of length n.
0xa3,0xa3,...	type[m,n]	2D array of shape m x n.
0xa3,0x28,...	type[n..m]	1D array with subscripts n through m inclusive (lenth m-n+1).
0xa3,0x28,0xa3,0x28,...	type[n..m,i..j]	2D array with subscripts n through m and i through j.
0xac,...	type[tagpath]	1D variable length array
0xac,0xa3,...	type[tagpath,n]	2D variable length array, with inner dimension length n.
0xac,...	type[*]	1D variable length array, autosized to buffer.
0xac,0xa3,...	type[*,n]	2D autosized array, with inner dimension length n.

In the above table, if "type" contains multiple keywords, enclose in parentheses.

Note that square brackets *contain* array *subscripts* in tagpaths, while they *define* array *dimensions* in token strings.

8.4.4 Structure Definition Segments

The CIP Specification has three type codes for use identifying/defining arbitrary structure types. Type code 0xa0, the abbreviated form, carries a 16-bit handle that is intended to uniquely identify a data type, by computing a CRC value over its definition. Type code 0xa2, the formal definition, is followed by a byte length of a nested list of data definition segments. The CIP spec includes a CRC algorithm, defined to use the formal definition as its source material.

The specification's third type code, 0xa8, the formal **handle** definition, is not implemented in this module.

With a formal definition, the member definitions are in packed form, even if the formal definition itself is in a padded context.

This module repurposes eight type codes as private extensions to support:

- Identifying a structure by name instead of 16-bit handle,
- Allowing longer formal definitions via a 16-bit length,
- Optional member names, via ansi or symbol segments preceding the member data type,
- and distinct member alignment options, for generic and Omron CJ and for Logix-style rules.

The token string forms are enclosed in parentheses if there is any ambiguity. The following combinations are supported:

Type Code	Syntax	Functionality
0xa0	struct n	Structured type identified by handle/CRC. "n" must decode to an integer.
0xae	s	Structured type identified by name, where name "s" is not a token keyword and is unambiguously an identifier.
0xae	struct s	Structured type identified by name, where "s" is arbitrary. Use when a structure name matches a primitive type or another keyword, or has punctuation. Use quotes if spaces or punctuation would make it ambiguous.
0xa2	(type type[n]...)	Structured type with anonymous members, up to 255 encoded bytes, single byte alignment.
0xa4	(type mname=type[n]...)	Structured type with optionally named members, up to 64k encoded bytes, single byte alignment.
0xa5	(align2 type mname=type[n]...)	Structured type like 0xa4, but with 16-bit maximum alignment.
0xa6	(align4 type mname=type[n]...)	Structured type like 0xa4, but with 32-bit maximum alignment.
0xa7	(align8 type mname=type[n]...)	Structured type like 0xa4, but with 64-bit maximum alignment.
0xa9	(align2c type mname=type[n]...)	Structured type like 0xa4, but with 16-bit Omron CJ forced alignment.
0xaa	(align4l type mname=type[n]...)	Structured type like 0xa4, but with 32-bit Logix-style alignment.
0xab	(align8l type mname=type[n]...)	Structured type like 0xa4, but with 64-bit Logix-style alignment.

When parsing, consecutive data definition segments without other delimiters will be wrapped in a single formal structure definition, as if wrapped in parentheses.

With type codes 0xa5 through 0xa7, the data alignment of each member is the lesser of the max alignment and that member's native data alignment. A nested structure's "native" data alignment is its declared alignment. Arrays are aligned according to their element's alignment.

With type code 0xa9, all members are force-aligned to 16-bit boundaries. (Omron CJ format).

With type codes 0xaa and 0xab, the native data alignment of members is only used for non-array elementary types. All other members use the structure's declared alignment. This "selective forced alignment" is the normal behavior of Logix processors.

Application notes:

- The keyword "align1" is optional for type codes 0xa2 or 0xa4. It is omitted when converting to string format.

- Logix processors prior to firmware v27 used 32-bit forced data alignment for all arrays and nested structures. Booleans are not permitted by themselves in Logix processors—they must be "hosted" in another member. Rockwell's software constructs hidden 8-bit members to host a named boolean, and packs consecutive named booleans together eight at a time.
- Logix processors from v27 onward use 64-bit forced data alignment for arrays and nested structures when any 64-bit type is a member, at any nesting depth.
- Omron's NJ structure types use the natural alignment of their members, up to 64-bit, then pad the structure to the largest alignment of it the members. Which then becomes the structure's natural alignment when nesting in other structures. Named booleans in NJ structures occupy and align to a complete 16-bit word of their own.
- Omron's CJ structure types use only 16-bit alignment, even for bytes and strings. Consecutive named booleans are packed into 16-bit words together, and in this library, must be hosted in a 16-bit word.

All of the aligned structures' data will be padded at the end to the structure's declared alignment.

8.4.5 Indirect Reference Segments

Structures may contain named members that actually refer to data within another member of the same structure. Most commonly, this is used for boolean values that are "hosted" in a member of an integral or bitstring type. More generally, an alias may be used to point arbitrary data types within another member that is a nested structure type. An alias may also be a tag itself, pointing to arbitrary data within another tag. The following encodings are available:

Type Code	Syntax	Alternate Syntax	Functionality
0xad	hbit h.b	hostedby h.b	Boolean in host member "h" bit number "b". The bit number may be 0-255 and is encoded in the following byte. The host member is encoded following that, as either an ansi symbol segment, a member segment, or a member number segment.
0xaf	aliasof tagpath		Any arbitrary data type determined by following the given tagpath. When present within a structure definition, the first element of the given tagpath must be a symbol or member segment that identifies a peer member of the structure, which must be a nested structure and must contain the balance of the tagpath. When used as a tag definition, its tagpath options are implementation defined.

8.5 Alternate Syntax

The string parser for element paths is shared for node addresses, route paths, and application paths. While most user interfaces display element paths using the parenthesized tokenpath & unparenthesized tagpath format described above, I/O module route and application paths in the Host Driver's Scanner default to bare token paths. All user interfaces will use implied parentheses on a bare token path as needed, and will attempt to decode hexadecimal "Padded Epaths" (nothing but pairs of hex digits and optional whitespace between pairs). The latter is particularly useful when copying application paths from EDS files.

8.6 Testing Element Paths

The parsing logic described above is available in Jython scripting contexts as the `system.cip.Path`. The static and instance methods available on `Path` are documented for the underlying library's `CipPath` Class. Similarly, decoding logic from the `PathSegment` Class is exposed as `system.cip.Segment`.

Use this example in the designer's script console to get started:

```
from system.cip import Path

def doPath(tagpath):
          p = Path.parse(tagpath)
          print "Normalized:", p
          for i, seg in enumerate(p):
                    print "%4d: %s" % (i, seg)
          print "Packed Encoding:", p.hexString(False)
          print "Padded Encoding:", p.hexString(True)

doPath("class 1 instance 1 attribute 1")
doPath("someTag.someMember[2].15")
doPath("arrayTag[2,3,4].someMbr.2")
doPath("(class 0x305 instance 0x114 attribute 0x64 A=DINT B=INT[3])B[0]")
```

Figure 25: Path Segment Parsing Test Script

The script above yields this:

```
>>>
Normalized: identity attr 1
   0: cls 1
   1: inst 1
   2: attr 1
Packed Encoding: 2001 2401 3001
Padded Encoding: 2001 2401 3001
Normalized: someTag.someMember[2].15
   0: ansi 'someTag'
   1: ansi 'someMember'
   2: memb 2
   3: bit 15
Packed Encoding: 9107736f6d65546167 910a736f6d654d656d626572 2802 3c030f
Padded Encoding: 9107736f6d6554616700 910a736f6d654d656d626572 2802 3c030f00
Normalized: arrayTag[2,3,4].someMbr.2
   0: ansi 'arrayTag'
   1: memb 2
   2: memb 3
   3: memb 4
   4: ansi 'someMbr'
   5: bit 2
Packed Encoding: 91086172726179546167 2802 2803 2804 9107736f6d654d6272 3c0302
Padded Encoding: 91086172726179546167 2802 2803 2804 9107736f6d654d627200 3c030200
Normalized: (cls 773 inst 276 attr 100 A=DINT B=INT[3])B[0]
   0: cls 773
   1: inst 276
   2: attr 100
   3: A=DINT B=INT[3]
   4: ansi 'B'
   5: memb 0
Packed Encoding: 210503 251401 3064 a40d00910141c4910142a303000000c3 910142 2800
Padded Encoding: 21000503 25001401 3064 a40d00910141c4910142a303000000c3 91014200 2800
>>>
```

Figure 26: Path Segment Parsing Test Output

9 Interpreting EDS Files

The example I/O Module Scanner configured within a Host Driver in Figure 16 through Figure 18 was created with a real 1734-OE2V Point I/O™ module and the EDS file: **0001007300380300.eds**. Note the '00' in place of the minor version. Electronic Data Sheets are organized into sections marked by a section name in square brackets, followed by lines containing "key=value;" pairs. The "value" part may be composed of multiple comma-separated fields and may span multiple lines. Indentation of keys and values is optional but typical, along with extra whitespace. Comments are marked with dollar signs, and may appear anywhere outside quoted strings, even in between various fields of a value. Rockwell's guidance to hardware developers on how to produce EDS files provides additional insight on how to interpret their content.

9.1 Identity

The electronic keying information is obtained from the [Device] section, as shown in the excerpt in Figure 27. This is required to be near the top of the file. In this section, each key has a single numeric or string value.

```
[Device]
          VendCode = 1;
          VendName = "Rockwell Automation/Allen-Bradley";
          ProdType = 115;
          ProdTypeStr = "Rockwell Automation Miscellaneous";
          ProdCode = 56;
          MajRev = 3;
          MinRev = 3;
          ProdName = "PointIO 2pt 24Vdc Analog Voltage Output";
          Catalog = "1734-OE2V/C";
          Icon = "1734yellow.ico";
```

Figure 27: EDS Device Section

The public parameters of the device will be described in the [Param] section. These parameters do not have any direct impact on the configuration described above, but will describe the values needed in the actual configuration tag and output tag, and the values to expect in the input tag.

9.2 Connection Options

Other than the route path, the information needed to complete the Connection Details scanner configuration is found in the [Connection Manager] section of an EDS file. An excerpt from the OE2V EDS file is shown in Figure 28, describing the Exclusive Owner connection to the OE2V.

```
[Connection Manager]
          Connection7 = $ Exclusive Owner (Direct to Module)
                    $ Catalogs 1734-OE2C, OE2V
                    $ Config Assem 123
                    $ Consume Assem 102
                    $ Produce Assem 101
          0x04030002, $ trigger & transport
          0x44240405, $ point/multicast & priority & realtime format
          ,,Assem126, $ O=>T default,description
          ,,Assem127, $ T=>O default,description
          ,Assem124, $ config part 1
          ,Assem123, $ config part 2
          "Direct Exclusive Owner", $ connection name
           "Direct Exclusive Owner Connection - Output data controls the state
                    (analog level) of each of the outputs. Input data contains
                    status of each of the outputs.", $ Help string
          "20 04 24 7B 2C 66 2C 65";
```

Figure 28: Connection Options

The first field of a "ConnectionN" key's value is its Trigger & transport support word. It should be broken down by byte to determine the transport class and data production triggers the module uses. The high byte must be one of the following values to be supported by this driver:

- 0x01 = listen only
- 0x02 = input only
- 0x04 = exclusive owner
- 0x08 = redundant owner

The second byte, little-endian bits 16 through 23, indicates which trigger types are supported. Bit 16 = 1, indicating "Cyclic" triggering, must be turned on. Support for Change-of-State triggering, indicated by bit 17 = 1, is planned.

The remaining two bytes, bits 0 through 15, indicate which transport classes are supported. Bit 1, for Transport Class 1, must be turned on.

The next field is the connection parameters support word, and must also be broken down. The high byte indicates the packet priorities supported in each direction. At least bits 26 and 30 must be turned on, indicating support for scheduled priority in both directions.

The next byte of the connection parameters indicates the support for multicast versus unicast in each direction. Bits 21 & 22 are the input direction's multicast and unicast support, respectively. Bits 17 & 18 are the output direction's multicast and unicast support. If both bits are turned on for a given direction, you may choose either packet type for that direction. The `1734-0E2V` module only supports multicast input and unicast output.

The next byte indicates the format used in the packet for transferring run/program mode to or from the target device. The high nibble, bits 12-14, indicate the input RT format, and the low nibble, bits 8-10, indicate the output RT format. Values supported by this driver are:

- 0x0 = Modeless
- 0x1 = Zero length idle
- 0x3 = Heartbeat
- 0x4 = 32-bit Run/idle header

Note that when the 32-bit run/idle header is specified, the actual data payload has four extra bytes.

The final byte of the connection parameters support word must have bits 0 and 2 turned on, indicating support for fixed payload sizes in both directions.

9.3 Connection Data

Completing the Connection Data section of the scanner configuration uses the balance of the "ConnectionN" key combined with key=value data from the [Assembly] section of the EDS file.

```
[Assembly]
          Assem102 = "Data","",
                     ,0x0000,,,
                     16,Param47,
                     16,Param48;
          Assem126 = $ O2T (output data)
                     "Ouput Data Assembly",
                     , $ Path, $ Length in bytes
                     , $ descriptor
                     ,, $ reserved
                     ,Assem102; $ Assembly sent to module
```

Figure 29: Scanner Output Assemblies

The next three fields of the "ConnectionN" key specify the output RPI (in microseconds), output size (in bytes), and output data format for the connection. For the `1734-0E2V`, the required RPI is blank, meaning no restriction. The size is also blank, meaning to use the size of the format argument. The format indicates the single parameter or the assembly of parameters that define the content of the output buffer. For the `1734-0E2V`, that is a reference to assembly specification #126, which in turn references assembly spec #102, as shown in Figure 29. The critical information is that the output buffer is a total of 32 bits long, 16 for Param47, and 16 for Param48. The imported L5X for this module tells us that these are Ch0Data and Ch1Data, respectively. Since the size of the assembly matches the size of the tag, no "Output Bytes" override is needed.

Similarly, the next three fields specify the input RPI (in microseconds), input size (in bytes), and input data format for the connection. For the `1734-0E2V`, the input format refers to assembly spec #127, which defines 32 bits of fault information, then refers to assembly #101. Assembly spec #101 defines two bytes of channel status information. As shown in Figure 30, total size of the input buffer is six (bytes). The input tag imported with the L5X is padded to eight bytes, so an "Input Bytes" override is needed.

```
[Assembly]
          Assem101 = "Status","",
                     ,,,,
                     8,Param49,
                     8,Param50;
          Assem127 = $ T2O (input data)
                     "Input Data Assembly",
                     , $ Path
                     , $ Length in bytes
                     , $ descriptor
                     ,, $ reserved
                     32,, $ Fault bits
                     ,Assem101; $ Assembly passed from module
```

Figure 30: Scanner Input Assemblies

The next four fields specify the size and format of the two possible configuration data blocks that would be

```
[Assembly]
          Assem124 = $ Private config in Forward Open, Module
                     "3rd Party Private Assembly",
                     , $ Path
                     , $ Length in bytes
                     , $ descriptor
                     ,, $ reserved
                     16,Param101, $ configuration revision
                                  (only valid value is 1)
                     16,; $ reserved - zeros
```

Figure 31: Scanner Configuration Assemblies

included in a Forward Open request message. The first data block is used with chassis-based modules, and is consumed by the chassis. The second data block is the configuration for the module itself. An L5X import general omits the first data block. For the `1734-0E2V`, the formats are given by assembly spec #124 and spec #123, respectively. The two values required by assembly spec #124 are constants, independent of the module configuration. These are placed at the end of the application path, as shown in Figure 18.

The final field in a "ConnectionN" value is the encoded form of the application path, not including the static config data segment. This should be entered into the Application setting by itself (to decode it), then edited as needed.

9.4 Assembly Structures

Unlike actual user-defined data types in Logix controllers, the I/O and configuration assemblies of an imported I/O module can be badly scrambled by missing information. An L5X export file may not provide all needed details to create the data types for the module. Rockwell's processors and I/O modules hide some data bytes and words from the user, and often skip bits in structures. So the data type "imported" for a module can be substantially shorter than the module expects, and named bits may not be in the right place. After an XML import, the imported data types (input, output, and configuration) should be compared with the assembly definitions in the EDS file to ensure the byte placement and bit definitions match.

10 Scripting Features and Functions

Several functions are registered under system.cip.* in Ignition's Script Manager, along with a variety of internal classes that would otherwise be hidden by Ignition's isolating classloaders. Among the internal classes exposed are a nearly-complete set of data types from the CIP Specification, not just the handful of types supported by the Logix tag and data type emulation described in Configuration. These data types are used directly to help encode and decode CIP message/reply payloads, and can be wrapped for efficient access to their content in jython.

10.1 Custom Jython Code Modules

Each device on the module's virtual backplane has an associated Jython code module, in Gateway scope. This code module's content is saved in and initialized from the configuration XML file, and can contain any valid jython code. Take care to properly XML-escape jython punctuation when editing XML directly, particularly the greater-than and less-than signs.

Figure 32: system.cip.getDevice(name=None, slot=-1, id=0)

Retrieves a Jython module object with the specified device's named Logix Tags as module-level globals, along with any other functions, classes, and variables from the XML configuration. Access to these variables, including assignment, is diverted to a Jython wrapper for the tag's datatype.

Argument	Data Type	Description
name	String	OPC Server Device Name
slot	Integer	Virtual Backplane Slot #
id	Long	OPC Server Device ID

Keyword-style invocation is allowed. If multiple criteria match, Name takes precedence over Slot, and Slot takes precedence over ID. 'None' is returned if no device matches.

Like other script modules, the code is executed once upon device startup, and again when any shared script module is edited. Any functions, classes, and module global variables declared in the jython code persist until the device is shut down. These code modules have a special module dictionary that automatically includes the Logix-style tags as module-level global variables, so long as the tag's name is a valid python name (no colon). This special dictionary, accessible via python's standard `globals()` function, also exposes a `subject` property pointing at the virtual device.

These module-level globals for the Logix-style tags wrap the raw data in special jython data types that allow Logix-like syntax. Arrays become jython "jarray" lists, structured types become objects with named properties, and the various integer types allow bit-wise access. Most importantly, use of these properties on the right hand side of a python expression retrieves the live value from the virtual module's tag, and assignment to these properties (left hand side) writes back into the virtual module's tag.

```
# Gateway event script
plc = system.cip.getDevice("MyDevice")

# Emulate ladder logic, like the follow Logix rung:
# XIC(MyTag[3].ItemZ.2)OTE(OtherTag[0].0)

# Note the underscore before constant bit numbers
plc.OtherTag[0]._0 = plc.MyTag[3].ItemZ._2
# Bits can also be indirect, but leave out the dot
plc.OtherTag[0][0] = plc.MyTag[3].ItemZ[2]
```

Figure 33: Logix Tags in Gateway Scripts

These jython wrappers for the device's tag data each also expose a `subject` property pointing at the tag object. This can be uses to inspect other attributes of the tag.

The syntax of these jython wrappers is intended to be as similar as practical to the syntax used within an Allen-Bradley Logix processor's code. The only major deviation is the treatment of bit access, both indirect and as constant bit numbers. See the example in Figure 33. Note that when assigning to one of these tag names within a function or class method, without any array subscript or dot-element, python's scope rules will create a function-local variable instead of assigning to the CIP data. Use python's **global** keyword at the beginning of the function to avoid this behavior.

10.2 Jython Data Events

Tags and Assemblies may provide a function name that will be called when a connection is made, Fresh data is written, any data is consumed, and/or data will be transmitted. Only ordinary functions defined within the virtual device's code module may receive these events, though that event function may call a function defined elsewhere in gateway scope, like a shared script module.

These events pass a single event object to the handler function, detailed in Figure 34.

The subject can be examined to determine the true source of an event when a handler function is shared by multiple objects. If the event logic needs to write back to the subject, consider using a dedicated handler and write to the module global that corresponds to the event source.

Each event contains a snapshot of the data in the subject from the time of the event, before the event is queued to the thread pool. Use this snapshot for any historical logging operations to avoid losing data. Event processing can be delayed for a variety of reasons.

The handler property contains the string name of the handler function. Used by the thread pool but available to the callee.

The optional detail property has additional information for connection actions on the subject. See Figure 35.

When the CipPyEvent logger level is set to Trace, every event processed will have an execution time report, with the time spent creating the snapshot and the time spent waiting in the background queue broken out.

Figure 34: CipPyEvent

Constructed by data objects within virtual devices when data traffic is occurring or connection status is changing.

Property	Data Type	Description
t_stamp	Date	Millisecond-resolution timestamp when the event occurred.
constructed	long	Nanosecond-resolution timestamp (from System.nanoTime) when the event was queued.
started	long	Nanosecond-resolution timestamp when the event execution started. If the delta between constructed and started is greater than the RPI involved, the event script should abort early.
subject	CipObject	The connectable data object that created the event.
detail	Boolean or CipPyCxEvent	For consume events, a boolean indicating a "fresh" packet, one with a changed 16-bit CIP serial number in the packet header. A nested object with details of connection status changes for connection handler events. *None* otherwise.
snapshot	PyObject	A wrapped copy of the data from the time the event occurred. Use this for data storage instead of accessing the live subject.

Figure 35: CipPyCxEvent

Constructed by data objects within virtual devices when connection status is changing during normal operation. Disconnect and Unsubscribed events may be dropped during virtual device shutdown or reconfiguration.

Property	Data Type	Description
type	String	"connect", "connectFail", "disconnect", "subscribe", or "unsubscribe".
object	varies	The AppOwner or AppConsumer that is (dis)connecting or (un)subscribing, respectively.
params	CxParams	Network Connection Parameters requested when connecting or subscribing. *None* otherwise. As described in the CIP Specification under Connection Manager Object Specific Service Parameters.
dirtrigcls	byte	The combined direction, trigger, and transport class code requested. As described in the CIP Specification under Connection Object Instances, Attribute #3.
e	Exception	Present on failure reports. *None* otherwise.

The execution time reports can also be generated for specific subjects by setting the subject's "Event Timing Debug" attribute to True, and setting the CipPyEvent logger level to Debug. Note that the event timing debug setting is not saved in the virtual device's XML, so will always be false at device or subject startup.

Warning: These module events use **unbounded queues** and deliver events one at a time per source object. It is vital that execution time within the event average less than the interval between events, or events will occupy substantial memory and eventually ***crash the gateway's JVM***. Event code should compute the nanosecond delta between `.constructed` and `.started` to see how long the event waited in queue. Old events should be discarded. Very high-speed operations should not perform time-consuming actions like SQL queries or network messaging within the event. Instead, construct a limited-size queue for the data to process, then process in bulk in a separate gateway event. Placing jython tuples into Java's LinkedBlockingQueue is recommended. Construct such queues with a **fixed capacity**.

Avoid using event handlers to execute state machines or perform handshaking that modifies I/O buffers for peer devices, especially if queuing. Instead, use a single gateway timer event at the appropriate pace (faster than RPI) to perform all such logic, as if it were a periodic task in a PLC.

10.3 CIP Messaging Access

A variety of CIP objects and services are not available as python data types, and no direct access to symbol data is available outside of gateway scope. The CIP specification includes many operations that have no OPC equivalents (at least, not yet in Ignition).

This module includes Jython functions and data types to construct virtually any desired CIP message request and parse the reply. Without the scanner option, these messages may be directed only at internal tags, symbols, and CIP objects, including via the backplane. When the scanner option is present, user-defined CIP messages may be sent through the scanner port to remote devices. Both Unconnected messaging and CIP Class 3 messaging connections are supported.

Figure 36: system.cip.getMsgPort(name=None, slot=-1, id=0, port=1)

Retrieves a local unconnected messaging port (in gateway) or messaging proxy (in client and designer scope) for a specific device on the virtual backplane.

Argument	Data Type	Description
name	String	OPC Server Device Name
slot	Integer	Virtual Backplane Slot #
id	Long	OPC Server Device ID
port	Integer	Port #1 is the backplane. TCP/IP port numbers are sequential starting with #2.

Keyword-style invocation is allowed. If multiple criteria match, Name takes precedence over Slot, and Slot takes precedence over ID. 'None' is returned if no device matches. The returned object has the following items:

Property	Data Type	Description
maxMsg	Integer	Maximum bytes allowed in an encoded request. Generally only meaningful on Class 3 buffered connections.
info	PortData	Only present in client/designer port proxies, contains information about the actual device found.

msgprocessor.send(request, nesting=0)

Argument	Data Type	Description
request	system.cip.Request	Or one of its subclasses, with an appropriate callback already added.
nesting	Integer	Number of target path segments to skip in the application path. Generally omitted.

Callbacks on requests must be implementations of system.cip.ReplyConsumer.

All access to CIP Messaging operations starts with an Unconnected Messaging manager object in a device's port. For inward-facing requests to the device itself, other than to communication objects, it doesn't really matter which port is used, and the default (backplane, port #1) is reasonable. For outbound requests with a scanner feature code, it is most efficient to use the same port # as in the first port segment of the route to the target. For inward-facing requests to a specific communication object, the correct port number is required. In client/designer context, consider caching responses to getMsgPort() in a script module global to minimize round-trips to the gateway.

Figure 37: system.cip.Request(target, serviceCode, payload=None)

Constructor for a generic CIP request object.

The returned request object has the following properties and methods:

Property	Data Type	Description
payload	ByteBuffer	Retrieve the current payload as a new ByteBuffer or replace the current payload with the bytes remaining in the ByteBuffer assigned to it.
serviceCode	Integer	Read-only after the constructor. 0-127.
target	system.cip.Path	Read-only after the constructor.

request.putPayload(buffer)

Argument	Data Type	Description
buffer	ByteBuffer	Retrieve the current payload and write it to the given buffer.

request.addCallback(callback)
request.addCallback(position, callback)

Argument	Data Type	Description
position	Integer	Callbacks are ordered. Use 0 to place this callback first in the chain.
callback	ReplyConsumer	The accept() method will receive the Reply object.

At its most basic, a request is composed of an application path, a service code, and an optional raw payload of bytes for the service to use. The corresponding reply echoes the service code, has a status code and optional extra status words, and an optional raw payload of bytes. Raw payloads are accepted and delivered as ByteBuffers.

Since it can be tedious and error-prone to construct raw request payloads and parse raw reply payloads, a variety of CIP data types are provided with suitable ByteBuffer putPayload() and setPayload() methods. These CIP data types can be wrapped in the same Jython-friendly wrappers used in the gateway-scoped per-device code modules, via system.cip.toPy(). Also, some subclasses of system.cip.Request are provided for request routing, request grouping, multiple attribute reads, and Class 3 connection operations.

Before using a message processor's send() method, attach one or more callbacks to your request. The callback object must be a class that extends system.cip.ReplyConsumer and implements the accept() method with one argument, the reply. The send() method does not return anything. Timing is asynchronous, so it is possible for the reply to arrive in the callback (in another thread) before the send() completes in the original thread.

See the JavaDoc for the underlying library for details of specific objects.

11 Troubleshooting

11.1 OPC Tag Subscriptions

11.1.1 Stale Data from a Host or Target Driver

Any data in a virtual module that receives packets from a scheduled I/O connection, either as a passive I/O output, an active I/O (scanner) input, or as a consumed tag, will be delivered to any OPC subscriber when actual packets arrive. And only when packets arrive. If the connection is broken, the data will become stale. This is the intended behavior. If the connection is delivering packets at a rate different from the scan class, the data will be stale. If the RPI on the I/O connection is 100ms, the tags must be in a scan class with a direct rate of 100ms. Consider using dedicated scan classes just for these I/O tags. Do ***NOT*** use leased scan classes with I/O data.

Tag event order and data delivery order within Ignition's SQLTag system are not guaranteed. To mitigate this, this module guarantees that all data from a newly-arrived I/O packet will be in place before any affected OPC subscribers are updated, but the order the subscribers are updated is not under the module's control. Users who need to be sure a single packet's data is processed together should use a Data Event function within the virtual module's script module. If that's not practical, use a tag change event on ***one*** single tag within the packet, and use system.opc.readAll() to obtain the rest of the data within that event script. In a transaction group, use OPC Read mode and a single trigger.

11.1.2 Excessive Subscriptions

Each Host Device or Target Device uses a single Java thread to service all of its OPC subscriptions. The OPC browse functions expose multiple ways to address much of the data, including string conversions for various arrays, precision timestamp conversions, and individual bits of integral data types. Dragging and dropping complex data types or arrays from the OPC browser to the SQLTags tree can create more tags than expected or needed, and the high data rates common to Logix I/O will place a heavy load on the OPC subscription thread. This will show up on the "Threads" section of the Gateway Console webpage.

Users should ***prune*** the Gateway's Tags tree of unneeded OPC items after every drag & drop action, and be sure to update the scan class for any subscription items you keep. Generally, you would want to keep any items that you want to use as tag event triggers, and any items you will be passing to client displays. Items that would only be used within scripts using system.tag.read*() should not have OPC tags. Unlike a conventional OPC driver, the Host and Target drivers produce no actual network traffic with system.opc.read*().

11.2 Scanner Connection Errors

There are numerous possible reasons for a Scanner or Consumed tag connection failure. This module provides status information on each connection under the Scanner Manager object for the virtual device. Scanner Manager instances are numbered from 1-n for each I/O scanner connection, in the order defined in the running XML configuration, and for consumed tags with the tag instance number plus 1,000,000. The Tag Instances are visible by number under the Logix Symbol Manager object.

Entry Status is the primary indicator from the Scanner Manager for what its state machine is doing. This is generally patterned after the corresponding Logix GSV instruction module data. Some hexadecimal codes for aborted connections can occur blended together, shown below with '?' wildcards.

EntryStatus Decimal	EntryStatus Hexadecimal	Description, Possible Solutions
0	0x0000	Idle. Should be a transient status.
varies	0x10?1 0x14?1	Connection aborted. Input Tag error. FaultCode and FaultInformation may have additional detail.
varies	0x10?2 0x14?2	Connection aborted. Input Tag does not support Class 1 connections
varies	0x10?3 0x14?3	Connection aborted. Input Tag inner element unsupported.
varies	0x101? 0x141?	Connection aborted. Output Tag error. FaultCode and FaultInformation may have additional detail.
varies	0x102? 0x142?	Connection aborted. Output Tag does not support Class 1 connections. Create an appropriate output tag – emulated Logix Tags support Class 1 connections.
varies	0x103? 0x143?	Connection aborted. Output Tag inner element unsupported. Direct access to tags must point at the first byte.
varies	0x14??	Connection aborted. Neither Input Tag nor Output Tag configured.
5376	0x1500	Connection aborted. No route configured.
5889	0x1701	Explicit Connection Rejection. More information will be in the FaultCode and FaultInformation properties.
5890	0x1702	Implicit Connection Failure. More information will be in the FaultCode and FaultInformation properties.
5891	0x1703	Implicit Connection Input Destination Failure. More information will be in the FaultCode and FaultInformation properties.
5892	0x1704	Implicit Connection Output Source Failure. More information will be in the FaultCode and FaultInformation properties.
5893	0x1705	Implicit Connection State Failure.
8192	0x2000	Connecting, request preparation.
12288	0x3000	Connecting, request sent.
16384	0x4000	Connected. FaultCode and FaultInformation cleared to zero.
24576	0x6000	Connection Inhibited by attribute #1.
28672	0x7000	Connection Disabled by operating mode change or missing license option.

The `FaultCode` and `FaultInformation` attributes provide additional information about failures, following the CIP standard for general status and extended status, respectively. Note that the free packet capture tool *Wireshark* will annotate connection failure messages containing these message codes. Combinations commonly encountered include the following, with solution possibilities to consider:

FaultCode Dec, Hex	FaultInformation Dec, Hex	Description, Possible Solutions
1	256, 0x0100	Connection occupied or duplicate connection attempt. Possible driver memory leak.
1	259, 0x0103	Unsupported Class and Trigger combination. Verify EDS file information/compatibility.
1	262, 0x0106	Ownership conflict, usually another scanner is controlling outputs or has the primary connection to inputs.
1	264, 0x0108	Unsupported connection parameter, generic. Verify EDS file information/compatibility.
1	265, 0x0109	Unsupported connection size, generic. Check input tag size, output tag size, and size limit overrides to ensure they match the EDS file's assembly sizes.
1	272, 0x0110	Missing required configuration data block. Either a static data segment must be used in the application path, or a configuration tag set up, or both.
1	273, 0x0111	Unsupported Requested Packet Interval, generic. Verify that RPIs are entered in *microseconds* in the configuration and fit the EDS file limits.
1	275, 0x0113	Out of connection resources. Can be any device along the route, not just the target. Examine a packet trace to pinpoint the offending device. Reduce network load at that point.
1	276, 0x0114	Unsupported Electronic Key, Vendor or Product Code.
1	277, 0x0115	Unsupported Electronic Key, Device Type.
1	278, 0x0116	Unsupported Electronic Key, Revision.
1	279, 0x0117	Invalid input or output application path. Verify EDS file information. Possibly examine packet captures from working scanners.
1	280, 0x0118	Invalid configuration application path. Verify EDS file information. Possibly examine packet captures from working scanners.

FaultCode Dec, Hex	FaultInformation Dec, Hex	Description, Possible Solutions
1	281, 0x0119	Listen-only connection not possible without other owner. Establish a primary connection before a listen-only connection.
1	282, 0x011a	Out of connection resources at the target application (not the connection manager). Examine device documentation for load reduction possibilities.
1	293, 0x0125	Unsupported redundant owner request.
1	% = 294, 0x????0126	Invalid configuration size. Upper half of FaultInformation indicates the largest configuration size allowed, in # of 16-bit words.
1	% = 295, 0x????0127	Invalid output size. Upper half of FaultInformation indicates the largest output size allowed, in # of bytes.
1	% = 296, 0x????0128	Invalid input size. Upper half of FaultInformation indicates the largest input size allowed, in # of bytes.
1	297, 0x0129	Invalid configuration application path. Verify EDS file information. Possibly examine packet captures from working scanners.
1	298, 0x012a	Invalid output application path. Verify EDS file information. Possibly examine packet captures from working scanners.
1	299, 0x012b	Invalid input application path. Verify EDS file information. Possibly examine packet captures from working scanners.
1	515, 0x0203	Connection Timed Out. Verify UDP network packets can pass in both directions.
1	516, 0x0204	Unconnected Message Timed Out. Verify TCP network packets can pass in both directions.
1	769, 0x0301	Target Device Out of Buffer Memory. Reduce the network load on the device.
1	785, 0x0311	Invalid/Unavailable Port in Route. Verify the route path to the target device.
1	786, 0x0312	Invalid/Unavailable Address in Route. Verify the route path to the target device.
1	789, 0x0315	Invalid Segment in Route. Verify the route path to the target device.
1	796, 0x031c	Miscellaneous Failure. Refer to target device documentation for possible causes.
5	Any	Path destination unknown. Usually means the first port number in the route path is not a valid external port for the Host Device instance. This is most commonly caused by no entry in the Host Device's "Local Addresses" setting, or an IP address in that setting that doesn't belong to the gateway. See §4.1.1.
9	N	Data Segment content error. Configuration rejected due to invalid value in 16-bit word #N (zero-based) of the data segment.
12, 0x0c	any	Target application object state error. Refer to target device documentation for possible causes.
16, 0x10	any	Target device state error. Refer to target device documentation for possible causes.
255, 0xff	any	General failure. Refer to target device documentation for possible causes.

12 Allen-Bradley Logix Firmware Variations

Rockwell Automation's Logix™ family of processors have changed greatly over the years since they were first introduced. Those changes are particularly important when emulating the tags/types of any Logix processor, as this driver's Host Device does. Exhaustive probing of various models with all available firmware versions has yielded a comprehensive index of those versions.

12.1 Elementary Data Types

Elementary data types, also called "primitive" data types in programming languages, are the fundamental units of data in these PLCs. Arrays and structures are composed of these types. All of these types are described in the CIP Specification, Volume 1, § C-2.1.1, though Rockwell Automation chose to use different names for some of them.

All Logix processors have the following elementary types: BOOL, SINT, INT, DINT, and REAL.

All Logix processors since firmware version 16 have the elementary type LINT.

Logix processors in the 5380, 5480, and 5580 families (like the 1756-L81) have the following elementary types, starting with firmware version 32: USINT, UINT, UDINT, ULINT, and LREAL.

Logix processors in the 5380, 5480, and 5580 families have the following CIP elementary types, starting with firmware version 34 (Logix names in parentheses): UTIME (DT), STIME (LDT), NTIME (LTIME), LTIME (TIME), and FTIME (TIME32).

Note that "LTIME" has a conflicting meaning between the two conventions. "DT" is also defined differently by the CIP specification, and is supported by this driver in the Client device type to handle CIP standard messaging. This driver accepts the Rockwell names when importing an L5X file. At all other points, the driver requires the CIP names.

12.2 Predefined Structured Types

The Logix Data Access manual prescribes how an external system should browse the tags and structured data types used in a Rockwell PLC. In L55, L6x, L7x, and corresponding CompactLogix models, many predefined structured data types show up in the browse protocol with different names from what a programmer sees within RSLogix 5000 or Studio 5000, and different from what shows up in an L5X project export. (In some cases, this appears to be as simple as spelling errors in the firmware.) This driver's Host Device type must do the same when emulating these processors. In current hardware, L8x and corresponding CompactLogix models, the browse protocol shows the same names as the programming tools and their project exports.

In the table below, "Usable" indicates that the structure can be used as the datatype for a simple controller tag. Some datatypes have further restrictions with arrays, program tags, and/or nested structure member usage. Certain datatypes, that show only as "present" in all models and versions, can appear is data types supplied by I/O modules.

L55 controllers support firmware versions 10 through 16. L6x controllers support firmware versions 12 through 20. L7x controllers support firmware versions 19[3] and higher. L8x controllers support firmware versions 29 and higher.

Type Name	ID	L55, L6x, L7x Families			L8x Family	
		Browse Protocol Name	Present	Usable	Present	Usable
ALARM	0x0f8b			All		All
ALARM_ANALOG	0x0ffa			v16-v35		All
ALARM_CONDITION	0x0f53				v31-v35	

3 Only in L72 processors. Others start at v20.

Type Name	ID	L55, L6x, L7x Families			L8x Family	
		Browse Protocol Name	Present	Usable	Present	Usable
ALARM_DIGITAL	0x0ffb			v16-v35		All
ALARM_SET	0x0f4f				v31-v35	
ALARM_SET_CONTROL	0x0f46			v32-v35		v32-v35
AUX_VALVE_CONTROL	0x0f68	AVC_STRUCT		v17-v35		All
AXIS_CIP_DRIVE	0x0ff9	AXIS_CIP		v18-v35		All
AXIS_CONSUMED	0x0fcb			All		All
AXIS_FEEDBACK	0x0fcc		v10-v29		v29	
AXIS_GENERIC	0x0fc9		v10-v11	v12-v35	v29-v35	
AXIS_GENERIC_DRIVE	0x0fcd			v15-v35	v29-v30	v31-v35
AXIS_SERVO	0x0fc8			All	v29-v30	v31-v35
AXIS_SERVO_DRIVE	0x0fc7	AXIS_SERVODRIVE		All	v29-v30	v31-v35
AXIS_VIRTUAL	0x0fca			All		All
BRKPT	0x0fcf		v10-v29		v29	
BUS_OBJ	0x0f35					v33-v35
CAMSHAFT_MONITOR	0x0f5f	CSM_STRUCT		v17-v35		All
CAM	0x0f88	PCAM		All		All
CAM_EXTENDED	0x0f19					v34-v35
CAM_PROFILE	0x0f89			All		All
CAM_PROFILE_EXTENDED	0x0f18					v34-v35
CB_CONTINUOUS_MODE	0x0f71	CBCM_STRUCT		v17-v35		All
CB_CRANKSHAFT_POS_MONITOR	0x0f74	CPM_STRUCT		v17-v35		All
CB_INCH_MODE	0x0f73	CBIM_STRUCT		v17-v35		All
CB_SINGLE_STROKE_MODE	0x0f72	CBSSM_STRUCT		v17-v35		All
CC	0x0fd2	COORDINATED_CONTROL		v17-v35		All
CONFIGURABLE_ROUT	0x0f66	ROUT2_STRUCT		v17-v35		All
CONNECTION_STATUS	0x0f76	CONNECTION_STATUS_STRUCT	v15-v16	v17-v35		All
CONTROL	0x0f81			All		All
COORDINATE_SYSTEM	0x0ffc	MOTION_COORDINATE_SYSTEM		v12-v35		All
COUNTER	0x0f82			All		All
DATALOG_INSTRUCTION	0x0f5a	DATALOG		v21-v35		All
DCAF_INPUT	0x0f5b	DCAF_STRUCT		v20-v35		All
DCA_INPUT	0x0f70	DCA_STRUCT		v20-v35		All
DCI_MONITOR	0x0f60	DCM_STRUCT		v17-v35		All
DCI_START	0x0f61	DCSRT_STRUCT		v17-v35		All
DCI_STOP	0x0f65	DCS_STRUCT		v17-v35		All
DCI_STOP_TEST	0x0f64	DCST_STRUCT		v17-v35		All
DCI_STOP_TEST_LOCK	0x0f63	DCSTL_STRUCT		v17-v35		All
DCI_STOP_TEST_MUTE	0x0f62	DCSTM_STRUCT		v17-v35		All
DEADTIME	0x0f90			All		All
DERIVATIVE	0x0fb3			All		All
DISCRETE_2STATE	0x0f98			All		All
DISCRETE_3STATE	0x0f99			All		All
DIVERSE_INPUT	0x0f7c	DIN_STRUCT		v15-v35		All
DOMINANT_RESET	0x0faa			All		All
DOMINANT_SET	0x0fa9			All		All
DYNAMICS_DATA	0x0f50			v30-v35		v30-v35
EIGHT_POS_MODE_SELECTOR	0x0f6d	EPMS_STRUCT		v17-v35		All
EMERGENCY_STOP	0x0f7e	ESTOP_STRUCT		v15-v35		All
ENABLE_PENDANT	0x0f78	ENPEN_STRUCT		v15-v35		All
ENERGY_BASE	0x0fe0					v31-v35
ENERGY_ELECTRICAL	0x0fe1					v31-v35
EXT_ROUTINE_CONTROL	0x0fd1			v12-v35		All
EXT_ROUTINE_PARAMETERS	0x0fd0			v12-v35		All
FBD_BIT_FIELD_DISTRIBUTE	0x0fc0	FBD_BIT_FILED_DISTRIBUTE		All		All
FBD_BOOLEAN_AND	0x0f9a			All		All
FBD_BOOLEAN_NOT	0x0f9d			All		All
FBD_BOOLEAN_OR	0x0f9b			All		All
FBD_BOOLEAN_XOR	0x0f9c			All		All

Type Name	ID	L55, L6x, L7x Families			L8x Family	
		Browse Protocol Name	Present	Usable	Present	Usable
FBD_COMPARE	0x0fbc			All		All
FBD_CONVERT	0x0fbe			All		All
FBD_COUNTER	0x0fb8			All		All
FBD_LIMIT	0x0fc2			All		All
FBD_LOGICAL	0x0fbd			All		All
FBD_MASKED_MOVE	0x0fc1			All		All
FBD_MASK_EQUAL	0x0fc3			All		All
FBD_MATH	0x0fba			All		All
FBD_MATH_ADVANCED	0x0fbb			All		All
FBD_ONESHOT	0x0fb9			All		All
FBD_TIMER	0x0fb7			All		All
FBD_TRUNCATE	0x0fbf			All		All
FILTER_HIGH_PASS	0x0f9e			All		All
FILTER_LOW_PASS	0x0f9f			All		All
FILTER_NOTCH	0x0fa0			All		All
FIVE_POS_MODE_SELECTOR	0x0f7f	FPMS_STRUCT		v15-v35		All
FLIP_FLOP_D	0x0fa5			All		All
FLIP_FLOP_JK	0x0fa6			All		All
FUNCTION_GENERATOR	0x0f91			All		All
HL_LIMIT	0x0f92			All		All
HMIBC	0x0ff8	JOG	v21-v35			All
IMC	0x0fd3	INTERNAL_MODE_CONTROL		v17-v35		All
INTEGRATOR	0x0fa3			All		All
LEAD_LAG	0x0f8f			All		All
LEAD_LAG_SEC_ORDER	0x0fa1			All		All
LIGHT_CURTAIN	0x0f7a	LCUR_STRUCT		v15-v35		All
MAIN_VALVE_CONTROL	0x0f69	MVC_STRUCT		v17-v35		All
MANUAL_VALVE_CONTROL	0x0f67	MMVC_STRUCT		v17-v35		All
MAXIMUM_CAPTURE	0x0fb5			All		All
MESSAGE	0x0fff			All		All
MINIMUM_CAPTURE	0x0fb4			All		All
MMC	0x0fd4	MODULAR_MULTIVARIABLE_CONTROL		v17-v35		All
MOTION_GROUP	0x0ffd	GROUP		All		All
MOTION_INSTRUCTION	0x0f87	MOTION		All		All
MOVING_AVERAGE	0x0f93			All		All
MOVING_STD_DEV	0x0fb0			All		All
MULTIPLEXOR	0x0fa8	MULTIPLEXER		All		All
MUTING_FOUR_SENSOR_BIDIR	0x0f6a	FSBM_STRUCT		v17-v35		All
MUTING_TWO_SENSOR_ASYM	0x0f6c	TSAM_STRUCT		v17-v35		All
MUTING_TWO_SENSOR_SYM	0x0f6b	TSSM_STRUCT		v17-v35		All
ODOMETER	0x0fe2			v31-v35		v31-v35
OSCILLATOR	0x0fad		v10-v29		v29	
OUTPUT_CAM	0x0fc5			All		All
OUTPUT_COMPENSATION	0x0fc6			All		All
PATH_DATA	0x0f51			v30-v35		v30-v35
PHASE	0x0f77			v15-v35	v29-v31	v32-v35
PHASE_INSTRUCTION	0x0f75			v15-v35		All
PIDE_AUTOTUNE	0x0fb6	AUTOTUNE_PIDE		All		All
PID	0x0f84			All		All
PID_ENHANCED	0x0f94			All		All
POSITION_DATA	0x0f52			v30-v35		v30-v35
POSITION_PROP	0x0f95			All		All
PROP_INT	0x0fa2			All		All
PULSE_MULTIPLIER	0x0fac			All		All
P_ANALOG_FANOUT	0x0f3c					v33-v35
P_ANALOG_HART	0x0f1b					v33-v35
P_ANALOG_INPUT	0x0f42					v33-v35
P_ANALOG_INPUT_DUAL	0x0f31					v33-v35

Type Name	ID	L55, L6x, L7x Families			L8x Family	
		Browse Protocol Name	Present	Usable	Present	Usable
P_ANALOG_INPUT_MULTI	0x0f34					v33-v35
P_ANALOG_OUTPUT	0x0f41					v33-v35
P_BOOLEAN_LOGIC	0x0f38					v33-v35
P_COMMAND_SOURCE	0x0f43					v33-v35
P_DEADBAND	0x0f36					v33-v35
P_DISCRETE_4STATE	0x0f17					v35
P_DISCRETE_INPUT	0x0f45					v33-v35
P_DISCRETE_MIX_PROOF	0x0f15					v35
P_DISCRETE_N_POSITION	0x0f16					v35
P_DISCRETE_OUTPUT	0x0f44					v33-v35
P_DOSING	0x0f30					v33-v35
P_HART_CODE_DESC_STATUS	0x0f1a					v33-v35
P_HIGH_LOW_SELECT	0x0f37					v33-v35
P_INTERLOCK	0x0f33					v33-v35
P_INTERLOCK_BANK_STATUS	0x0f32					v33-v35
P_LEAD_LAG_STANDBY	0x0f23					v33-v35
P_LEAD_LAG_STANDBY_MOTOR	0x0f22					v33-v35
P_MOTOR_DISCRETE	0x0f2e					v33-v35
P_PERMISSIVE	0x0f3b					v33-v35
P_PID	0x0f24					v33-v35
P_PRESS_TEMP_COMPENSATED	0x0f3d					v33-v35
P_RESTART_INHIBIT	0x0f3e					v33-v35
P_RUN_TIME	0x0f40					v33-v35
P_STRAPPING_TABLE_ROW	0x0f39					v33-v35
P_TANK_STRAPPING_TABLE	0x0f3a					v33-v35
P_VALVE_DISCRETE	0x0f2f					v33-v35
P_VALVE_STATISTICS	0x0f3f					v33-v35
P_VARIABLE_SPEED_DRIVE	0x0f28					v33-v35
RAC_CODE_DESCRIPTION	0x0f2d					v33-v35
RAC_EVENT	0x0f2c					v33-v35
RAC_ITF_DVC_PWRDISCRETE_CMD	0x0f2b					v33-v35
RAC_ITF_DVC_PWRDISCRETE_SET	0x0f2a					v33-v35
RAC_ITF_DVC_PWRDISCRETE_STS	0x0f29					v33-v35
RAC_ITF_DVC_PWRMOTION_CMD	0x0f21					v33-v35
RAC_ITF_DVC_PWRMOTION_INF	0x0f20					v33-v35
RAC_ITF_DVC_PWRMOTION_SET	0x0f1f					v33-v35
RAC_ITF_DVC_PWRMOTION_STS	0x0f1e					v33-v35
RAC_ITF_DVC_PWRVELOCITY_CMD	0x0f27					v33-v35
RAC_ITF_DVC_PWRVELOCITY_SET	0x0f26					v33-v35
RAC_ITF_DVC_PWRVELOCITY_STS	0x0f25					v33-v35
RAMP_SOAK	0x0f97			All		All
RATE_LIMITER	0x0fb2			All		All
REDUNDANT_INPUT	0x0f7b	RIN_STRUCT		v15-v35		All
REDUNDANT_OUTPUT	0x0f7d	ROUT_STRUCT		v15-v35		All
SAFELY_LIMITED_POSITION	0x0f4a				v31-v32	v33-v35
SAFELY_LIMITED_SPEED	0x0f4c				v31-v32	v33-v35
SAFETY_FEEDBACK_INTERFACE	0x0f4e				v31-v32	v33-v35
SAFETY_MAT	0x0f6f	SMAT_STRUCT		v17-v35		All
SAFE_BRAKE_CONTROL	0x0f47				v31-v32	v33-v35
SAFE_DIRECTION	0x0f4b				v31-v32	v33-v35
SAFE_OPERATING_STOP	0x0f49				v31-v32	v33-v35
SAFE_STOP_1	0x0f4d				v31-v32	v33-v35
SAFE_STOP_2	0x0f48				v31-v32	v33-v35
SAWTOOTH	0x0fae		v10-29		v29	
SCALE	0x0f8a			All		All
SEC_ORDER_CONTROLLER	0x0fa4			All		All
SELECTABLE_NEGATE	0x0fb1			All		All
SELECTED_SUMMER	0x0fa7			All		All

Type Name	ID	L55, L6x, L7x Families			L8x Family	
		Browse Protocol Name	Present	Usable	Present	Usable
SELECT	0x0f8c			All		All
SELECT_ENHANCED	0x0f8d			All		All
SEQUENCE	0x0f5d			v28-v35	v30-v35	
SEQ_BOOL	0x0f54	SEQUENCE_BOOL		v28-v35	v30-v35	
SEQ_DINT	0x0f57	SEQUENCE_DINT		v28-v35	v30-v35	
SEQ_INT	0x0f56	SEQUENCE_INT		v28-v35	v30-v35	
SEQ_REAL	0x0f58	SEQUENCE_REAL		v28-v35	v30-v35	
SEQ_SINT	0x0f55	SEQUENCE_SINT		v28-v35	v30-v35	
SEQ_STEP	0x0f5c	SEQUENCE_STEP		v28-v35	v30-v35	
SEQ_STRING	0x0f59	SEQUENCE_STRING		v28-v35	v30-v35	
SEQ_TRANSITION	0x0f6e	SEQUENCE_TRANSITION		v28-v35	v30-v35	
SERIAL_PORT_CONTROL	0x0fc4	ASCIICONTROL		All		All
SFC_ACTION	0x0fde			v11-v35		All
SFC_STEP	0x0fdf			v11-v35		All
SFC_STOP	0x0fdd			v11-v35		All
SIGNED_ODOMETER	0x0fe3			v31-v35		v31-v35
SPLIT_RANGE	0x0f96			All		All
STRING	0x0fce	ASCIISTRING82		All		All
STRING_16	0x0f1d			v33-v35		v33-v35
STRING_32	0x0f1c			v33-v35		v33-v35
S_CURVE	0x0fab			All		All
THRS_ENHANCED	0x0f5e			v17-v35		All
TIMER	0x0f83			All		All
TOTALIZER	0x0f8e			All		All
TWO_HAND_RUN_STATION	0x0f79	THRS_STRUCT		v15-v35		All
UP_DOWN_ACCUM	0x0faf	UP_DOWNN_ACCUM		All		All

Notice that L8x family hardware, while introducing a large number of new datatypes for new features, does lack functionality available in the L7x family at matching firmware levels. The last being the sequence data types—still not supported in L8x at this time.