PYRAMID
www.pyramid.tech

IGX

IGX - Programmer Manual

Document ID: 4382294044

Version: v8

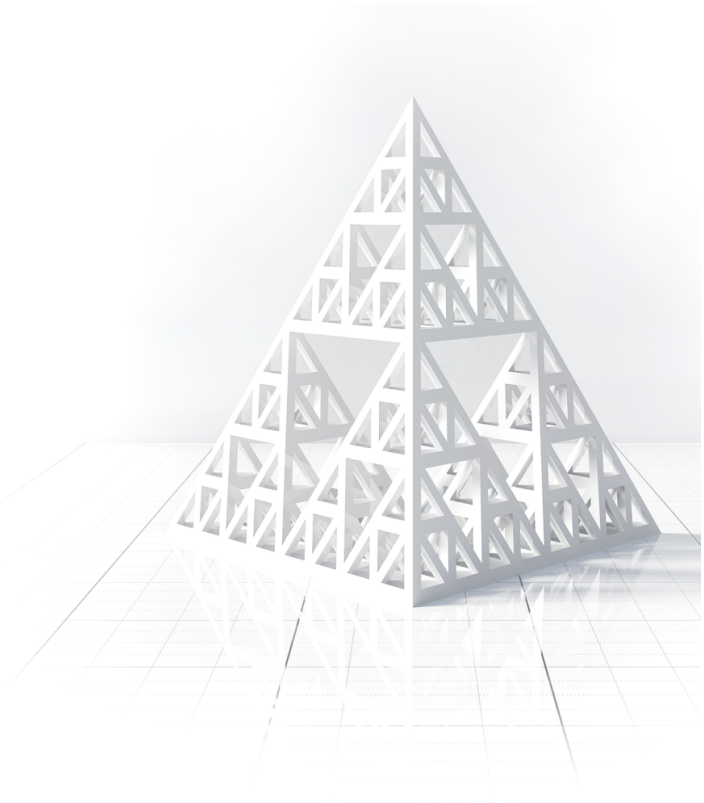


Table of Contents

1 Introduction 8
2 Document Control 9
2.1 Version History 9
2.2 Approvals 9
2.2.1 Signatures 9
3 IGX Network Protocols 10
3.1 Overview of IGX Protocols 10
3.1.1 One IO Tree, Every Protocol 10
3.1.2 HTTP Protocol 11
3.1.3 WebSocket Protocol 11
3.1.4 EPICS Protocol 11
3.1.5 SFTP Protocol 12
3.1.6 Qnet Protocol 12
3.1.7 Protocols IGX Uses to Reach Other Equipment 13
3.2 IGX HTTP Protocol Guide 13
3.2.1 IGX HTTP Server Introduction 13
3.2.2 HTTP Quick Start 14
Use cURL to GET field value. 14
use cURL to PUT a field value. 14
Use Microsoft Excel to GET field value. 14
Use Python and requests library to SET and GET field value 14
Suitable HTTP Libraries 14
3.2.3 The HTTP Protocol Basics 15
The TCP Connection 15
HTTP URLs 16
HTTP Request Structure 16
HTTP Response Structure 17
3.2.4 HTTP Python Examples 18
Python Requests Sessions 18
Sending a GET Request 19
Sending a PUT Request 19
3.2.5 Postman for HTTP Testing 21
Sending a GET Request 21
Sending a PUT Request 21
3.2.6 PLC HTTP Programming 22

Converting Strings to Numeric or Binary Values 23
Protocol Gateways 23
Custom Code Solutions 23
3.3 IGX WebSocket Protocol Guide 25
3.3.1 WebSocket Overview 25
3.3.2 Connecting 26
URL and Port 26
Subprotocol Negotiation 26
Keep-alive and Idle Timeout 27
Close Codes 27
3.3.3 IGX Message Protocol 28
Paths 28
Values and Timestamps 28
Config Event 29
Subscribe Event 30
Subscribe Table Event 31
Get Event 31
Set Event 32
Update Event 33
Update Table Event 34
Update ID Event 36
Get ID Event 37
Typical Protocol Sequence 37
3.3.4 MessagePack Encoding and Binary Blobs 38
Array Blobs 38
Buffered History Blobs 39
Table Row Blobs 39
3.3.5 Performance Guidance 39
3.3.6 Suitable WebSocket Libraries 40
3.3.7 Python T1 Example 41
3.3.8 High-Performance Python Client 41
3.3.9 Troubleshooting 45
3.3.10 Conclusion 46
3.4 IGX Apache Proxy Server Guide 46
3.4.1 Prerequisites 46
3.4.2 Setup Notes 46
3.4.3 Steps to Configure Apache Proxy 46
1. Edit the Apache Configuration File 46

2. Enable Required Apache Modules 47
3. Verify Configuration 47
3.4.4 Explanation of Configuration 48
ProxyPass and ProxyPassReverse 48
Dynamic Mapping with RewriteRule 48
3.5 IGX EPICS Protocol Guide 48
3.5.1 EPICS Overview 48
EPICS Network Protocols and Ports 49
Automatic Network Discovery 50
Using EPICS Across Network Barriers 51
EPICS CA Address Configuration 51
Choosing EPICS for Your Control System 52
3.5.2 IGX EPICS Interface 52
Handling Multiple IGX Devices 53
IGX PV Alias Map File 53
3.5.3 EPICS Utility Programs 55
Getting the Current Hostname 55
Setting a New Hostname 55
Monitoring Heartbeat 55
3.5.4 EPICS Python Examples 56
Python Library Overview 56
Getting a PV (IO) Value 57
Putting a PV (IO) Value 57
Zeroing a T1 Probe 58
Polling Heartbeat 59
Subscribing to Heartbeat 60
3.6 IGX SFTP Protocol Guide 61
3.6.1 Use SFTP for IGX IO Data 61
Accessing IGX IO Data via SFTP 61
Python Example 61
3.7 IGX Qnet Protocol Guide 62
3.7.1 Qnet Overview 62
Network Protocol 62
Device Discovery 63
The Net Directory 63
Network Packet Passing 63
File Exposure between Nodes 64
3.7.2 Integrating Qnet with IGX 64

Accessing IO Data through Qnet 64
File Path Conventions in IGX 64
3.7.3 Python Qnet Example 64
4 IGX Standard IO Interfaces 66
4.1 IGX Button IO Interface 66
4.1.1 Overview 66
4.1.2 Button Variants 66
4.1.3 Triggering a Button 66
4.1.4 Python HTTP Example 67
4.1.5 Hidden "Spinning" Indicator 67
4.1.6 Real-World Example: Data Acquisition Table Controls 67
4.2 IGX High Voltage IO Interface 68
4.2.1 Overview 68
Interface IO 68
Module States 68
Safety Interlocks 68
Internal Voltage Sense Circuit 69
External Voltage Sense Circuit 69
4.2.2 Use Case Examples 69
Setting and Enabling High Voltage Output 69
Reading Internal Voltage Values 69
Monitoring External Voltage Values (Optional) 69
4.3 IGX Relay IO Interface 70
4.3.1 Basic Relay Control IO 70
4.3.2 Advanced Control IO 70
4.3.3 Web GUI 70
4.4 IGX Offset Correction (Zero) IO Interface 71
4.4.1 Overview 71
Interface IO 71
4.5 IGX Dose Controller IO Interface 72
4.5.1 Overview 72
Primary Control IO 72
Calibration and Configuration IO 74
Q-Pulse Charge Monitor IO 74
Quadrant Detector IO 74
4.5.2 Controller Interlock and Permits 75
Ready Permit 75
Stopping Permit 75

Pausing Permit ... 75
4.5.3 Simple Charge Prescription Example ... 76
4.6 IGX Admin IO Interface ... 76
4.6.1 Overview ... 76
4.6.2 Permission Policy IO Interface ... 77
4.7 IGX Interlocks and Permits IO Interface ... 78
4.7.1 Overview ... 78
Common Interlock Control IO ... 78
InterlockIO Specializations ... 78
5 IGX Data Acquisition Table API Guide ... 80
5.1 The Capture State Machine ... 80
5.2 IO Reference ... 80
5.2.1 Status IO ... 80
5.2.2 Table Capacity and Memory ... 81
5.2.3 Configurable Parameters ... 82
5.2.4 Start Trigger and Stop Trigger ... 82
5.2.5 Control Buttons ... 83
5.2.6 Per-ADC Channel Selection ... 83
5.3 Capturing a Buffer Once ... 84
5.4 Continuous Acquisition with a Circular Buffer ... 85
5.4.1 Practical Considerations ... 87
6 IGX File Format Specifications ... 88
6.1 IGX JSON IO Files ... 88
6.1.1 Introduction ... 88
Example JSON ... 88
Field Types ... 88
Index JSON Files ... 89
Field JSON Files ... 89
JSON Schema ... 89
Availability of Protocols ... 90
Helpful Links ... 90

Table of Figures

Figure 1 - WebSocket Timing Diagram 25
Figure 2 - JSON Event Timing Diagram..... 38
Figure 3 - Example EPICS Timing Diagram 50
Figure 4 - The IGX PV XML file upload section..... 54
Figure 5 - FX4 Offset Correction 71

1 Introduction

Document ID: 2439249921

Author	Matthew Nichols
Purpose	Explain IGX programmer's concepts in order to facilitate the development of complementary applications.
Scope	IGX related software development outside the scope of IGX itself.
Intended Audience	Software developers interested in creating applications that work with the IGX control system framework.

2 Document Control

2.1 Version History

Version	Description	Saved by	Saved on
v8	Updated network protocols	Matthew Nichols	Nov 13, 2026 10:18 PM
v7	Updated WS protocol for tables with examples	Matthew Nichols	Jul 21, 2026 4:44 PM
v6	Added documentation for new EPICS features	Matthew Nichols	Dec 23, 2025 2:46 PM
v5	Added additional EPICS information	Matthew Nichols	May 14, 2025 4:10 PM
v4	Updated admin IO interface	Matthew Nichols	Mar 13, 2025 10:35 PM
v3	Added permission policy interface	Matthew Nichols	Jun 3, 2024 9:33 PM
v2	Improved network protocol documentation.	Matthew Nichols	Apr 10, 2024 9:54 PM
v1		Matthew Nichols	Feb 21, 2024 11:40 PM

2.2 Approvals

This document has been reviewed and approved as follows.

Document Control Not Reviewed

Current document version: v.1

No reviewers assigned.

2.2.1 Signatures

Friday, Nov 13, 2026, 10:22 PM UTC, (v. 1)
Matthew Nichols signed ; meaning: **Review**

3 IGX Network Protocols

3.1 Overview of IGX Protocols

IGX devices support multiple network protocols to enable integration with various applications and systems. Each protocol has its own strengths and weaknesses, and it is important to choose one that fits your particular application well. Here is an overview of the supported protocols and their usage in the context of IGX.

Writing a new client? Use WebSocket with MessagePack encoding. It is the only protocol that delivers buffered samples without loss and streams data acquisition tables, and it is the protocol the IGX web interface itself uses. Use HTTP for simple scripts and one-off reads or writes.

Protocol	Best for	Lossless buffered data	Client platform
WebSocket	Live monitoring and control, high-rate data, many IO over one connection, tables	Yes	Any
HTTP	Simple scripts, occasional reads and writes, file transfer	No	Any
EPICS	Integrating with an existing EPICS control system at 1 to 20 Hz	No	Any with EPICS Channel Access libraries
SFTP	Occasional secure file transfers	No	Any with an SSH client
Qnet	Code running on another QNX node on the same network	No	QNX only

3.1.1 One IO Tree, Every Protocol

Every protocol exposes the same tree of nodes and IO, addressed by path. A path names a node or IO from the root of the device, for example `/web_server/port` for the web server's port number. Each IO holds its current value in a child field called `value`. The table shows how each protocol addresses that same IO.

Protocol	Addressing `/web_server/port`
WebSocket	`/web_server/port/value` in `subscribe` and `set` messages
HTTP	`GET http://<device-ip>/io/web_server/port/value.json` returns `80`
SFTP	The file `/io/web_server/port/value.json`
Qnet	The file `/net/<node-name>/io/web_server/port/value.json`
EPICS	PV `web_server/port` or `web_server.port`, optionally prefixed with the device hostname or IP address (`<hostname>:web_server.port`), or an alias from the PV map

The `/io` directory is a virtual file system served by the running IGX application, so files under it are generated on demand from live data rather than stored on disk. Each IO field appears as `<field>.json` containing its JSON value; writable fields accept a new JSON value written to the same file. Each node also provides `index.json` (a full description of the node, its IO and their metadata), `children.json`, and `index.html`.

3.1.2 HTTP Protocol

All IGX devices are equipped with a high-performance HTTP file server that allows you to build external applications with ease. It listens on the same port as WebSocket (default 80, set by `/web_server/port`) and supports `GET`, `HEAD`, and `PUT`; other methods return `501 Not Implemented`.

For safety, only the `/io`, `/root`, `/tmp`, and `/api` directories are reachable, plus `/ram` when the RAM disk is mounted. Requests for any other path return `403 Forbidden`. Single-segment paths such as `/app.js` are served from `/root/web`, and `/` serves the device's web interface.

IO data is read and written through the `/io` virtual file system described above: prepend `/io` to the IO path and append `/value.json` to read or write a value, or `/index.json` to read a node's full description.

Strengths

- Very simple to implement and prototype.
- Robust and mature libraries.
- Practically universal language and platform support.
- Strong cross-network and internet support.

Weaknesses

- Some extra overhead per transaction: one request per file.
- Polling only, with no way to ask for just the values that changed.
- Lacks support for lossless buffered data.

See the full guide here: IGX HTTP Protocol Guide (see page 13)

3.1.3 WebSocket Protocol

WebSocket is a communication protocol that facilitates two-way, real-time communication between a client (typically a web browser) and a server via a single, long-lived connection. Designed to work over the same ports as HTTP, WebSocket easily integrates with existing web infrastructure.

IGX uses WebSocket for a polling subscription protocol. The client subscribes to a list of IO once, then sends `get` whenever it wants new data, and the device replies with only what changed since the previous poll. Buffered IO return every sample since the previous poll with nanosecond timestamps, and data acquisition tables stream their rows. Messages are encoded as JSON text or as MessagePack binary; MessagePack can also carry arrays and sample histories as raw binary blobs, which is the fastest option.

Strengths

- Enables streaming lossless buffered high-speed data.
- Can manage hundreds of IO through a single TCP connection.
- Compact binary encoding (MessagePack with raw array blobs) for maximum throughput.
- Strong community of developers and libraries.
- Strong cross-network and internet support.

Weaknesses

- Can be complex to implement without pre-made libraries.
- Requires code that manages the state of the connection and handles reconnect edge cases.
- The device closes connections that are idle for 10 seconds, so clients must poll or ping regularly.

See the full guide here: IGX WebSocket Protocol Guide (see page 25)

3.1.4 EPICS Protocol

The Experimental Physics and Industrial Control System (EPICS) is a popular, open-source, distributed control system used in scientific instruments like particle accelerators, telescopes, and

large-scale experiments. EPICS streamlines communication between hardware devices and software applications, enabling effective data acquisition, device control, and monitoring.

IGX devices include an EPICS Channel Access server (EPICS Base 3.16), active on devices whose configuration includes an `epics_server` node. Every IO is available as a PV named by its IO path, with `/` or `.` as the separator, optionally prefixed with the device's hostname or IP address and a colon. Friendlier PV names can be mapped to IO paths in `/root/config/process_variable_map.xml`.

EPICS is generally only suitable for low-speed data between 1 and 20 Hz. If your application calls for higher speed data, consider WebSockets instead.

Strengths

- Standard in some big-experiment facilities.
- Simple to use with the given libraries once everything is set up.
- PV names can be resolved on a local network without knowing IP addresses of devices up front.

Weaknesses

- No good way to send large amounts of high-speed data.
- Libraries are hard to compile for some platforms with lots of dependencies.
- Lack of broad software community support.
- Lots of pitfalls in the implementation process, which can lead to a frustrating development experience.
- Some network administrators block broadcast IP packets that EPICS relies on for discovery, rendering the automatic IP resolution feature useless.
- No good support for cross-network communication. Requires lots of careful setup with your network administrator and manually setting server IP addresses.

See the full guide here: IGX EPICS Protocol Guide (see page 48)

3.1.5 SFTP Protocol

Secure File Transfer Protocol (SFTP) is a robust and secure protocol for transferring files over a network. SFTP uses the Secure Shell (SSH) protocol for data encryption and authentication, making it a reliable choice for secure file transfers. By integrating SFTP with IGX systems, you can transfer IO data and other files while ensuring data confidentiality and integrity during transmission. IO data is read and written through the same `/io` files that HTTP uses.

SFTP is a good choice if you are already using SFTP transferring for other purposes, and you only need to query or upload a few files periodically from the IGX system. This protocol should not be used for data intensive applications where low latency is important.

Strengths

- Cryptographically secure and encrypted.
- Useful for all file transfers not just for IO.
- Strong cross-network and internet support.

Weaknesses

- High latency per transaction.
- Low overall network bandwidth capability.
- Lacks support for lossless buffered data.

See the full guide here: IGX SFTP Protocol Guide (see page 61)

3.1.6 Qnet Protocol

Qnet is a transparent distributed processing protocol native to the QNX Neutrino RTOS. It allows multiple QNX-based systems to communicate and share resources with one another as if they were part of a single, unified system. Qnet enables seamless access to files and processes across the network, making it an ideal choice for distributed applications and complex system architectures.

In the context of IGX devices, Qnet can be utilized for sharing IO data and accessing remote files or processes. Another QNX node reaches a device's IO through the same `/io` virtual file system, at `/net/<node-name>/io`, without any specialized API or custom communication protocol.

Qnet is a good choice if the system your code is deploying to is already a QNX device with Qnet enabled. For instance, if you are writing code that is intended to be embedded and run on a Pyramid device. If your code is running on a Linux or Windows system, Qnet is not recommended.

Strengths

- Great performance and latency.
- Easy to implement and use, if you are already using QNX.

Weaknesses

- Requires the QNX operating system.
- Highly dependent on custom software.
- Currently lacks support for buffered data.
- Lacks good cross-network or internet support.

See the full guide here: IGX Qnet Protocol Guide (see page 62)

3.1.7 Protocols IGX Uses to Reach Other Equipment

IGX also contains Modbus TCP, OPC UA, and Qnet clients, which an IGX device uses to read from and write to other equipment such as motor drives and PLCs. In these roles IGX is the client or master; they are not ways for external software to access an IGX device.

3.2 IGX HTTP Protocol Guide

3.2.1 IGX HTTP Server Introduction

All IGX devices include a lightweight HTTP/1.1[1] server that external applications can use to read and write IO values and selected files. It listens on port 80 by default (configurable at `/web_server/port`). Only these path roots are served: `/io/` (live IO data), `/api/`, `/root/`, `/tmp/`, and `/ram/` (when mounted). Requests outside these roots return `403 Forbidden`. A request for `/` serves the device web UI, and single-segment paths such as `/favicon.ico` are served from `/root/web`.

IO data lives in a virtual file system mounted at `/io`. Every IO field appears as a JSON file named `/io/<IO path>/<field>.json`, for example `/io/heartbeat/value.json` or `/io/heartbeat/label.json`. These files are generated on demand from live data, so every GET returns the current value. When accessing IO data through HTTP, prepend `/io` to the IO path and append the field file name (usually `/value.json`).

> HTTP is the simplest way to reach IGX and suits occasional reads, configuration writes, and integrations such as Excel, PLCs, and scripts. Each request reads or writes one field. To stream many values at high rates, use the IGX WebSocket Protocol Guide (see page 25) instead.

By leveraging the IGX HTTP server, developers can easily create web-based or traditional applications that interact with the control system. The HTTP server's high-performance capabilities enable it to handle a large number of requests simultaneously, making it an excellent choice for building responsive and dynamic applications. The use of plain text JSON[2] files for storing IO data makes it easy to integrate with various programming languages and data exchange formats.

1. https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol
2. https://en.wikipedia.org/wiki/JSON

3.2.2 HTTP Quick Start

To query an IO value field in IGX, the path should be in the following format: `/device/component/channel/value`. You can use various tools and programming languages to GET or PUT the value of an IO field. Here are some quick examples:

Use cURL[3] to GET field value.

```
curl -X GET http://<IP ADDRESS>/io/heartbeat/value.json
```

use cURL[4] to PUT a field value.

```
curl -X PUT -d '"MY-DEVICE"' http://<IP ADDRESS>/io/net/hostname/value.json
```

The body must be valid JSON, so string values keep their double quotes. The single quotes stop the shell from removing them.

Use Microsoft Excel to GET field value.

Enter the following function into a cell:

```
=WEBSERVICE("http://<IP ADDRESS>/io/heartbeat/value.json")
```

Click outside the cell and use `Ctrl` + `Alt` + `F9` to refresh the value.

Use Python and requests library to SET and GET field value.

```
import json
import requests

url = "http://<IP ADDRESS>/io/net/hostname/value.json"

# PUT replies 200 with an empty body, so do not call .json() on it.
requests.put(url, json.dumps("NEW-NAME"), timeout=2.0).raise_for_status()

# Read the value back to confirm the write took effect.
print(requests.get(url, timeout=2.0).json())
```

Suitable HTTP Libraries

In this guide we will be using Python[5] and the requests library[6] for example code, but any language and HTTP library should work well.

The following is a table of libraries that can be used with IGX for different languages. The list is incomplete, and there are many more options to choose from in addition to these.

Language	Library	Notes
Python	requests[7]	Very simple to use library, used at Pyramid for internal testing tools. Uses urllib3 under the hood.
Python	urllib3[8]	Lower-level library, useful if you want to have more control and don't want extra code.
JavaScript	fetch[9]	Newer built-in JS function for making HTTP requests.

3. https://curl.se/
4. https://curl.se/
5. https://www.python.org/
6. https://docs.python-requests.org/en/master/
7. https://pypi.org/project/requests/
8. https://urllib3.readthedocs.io/en/stable/

Language	Library	Notes
JavaScript	XMLHttpRequest[10]	The old way of making HTTP requests, useful if your environment doesn't support modern JavaScript.
C	libcurl[11]	A very complete library that includes HTTP and other file transfer protocols.
C++	cpr[12]	A C++ wrapper for libcurl that is inspired by the Python library requests.
C++	HTTPRequest[13]	Single header file implementation. Great for small projects or if you need a simple build process.
C#	HTTPClient[14]	Newer and preferred way to make HTTP requests in .NET.
C#	HTTPWebRequest[15]	The older way of making an HTTP request.
Java	Java 11 HttpClient[16]	Newer built-in library for HTTP.
Java	Apache HttpClient[17]	Older but still a very popular library.
LabView	GET VI[18]	Built-in LabView HTTP VI to send GET requests.
LabView	PUT VI[19]	Built-in LabView HTTP VI to send PUT requests.

3.2.3 The HTTP Protocol Basics

This section provides an overview of the basics of HTTP and its usage in the context of IGX. By understanding HTTP, developers can easily communicate with IGX and access the data generated by the control system using a variety of programming languages and tools that support HTTP communication. If you already are familiar with HTTP, feel free to skip this section and move on to the IGX specific examples.

We will cover the TCP connection required by HTTP, the structure of HTTP requests, URLs, and HTTP responses. Later, we also provided Python and Postman examples to help you understand how to use HTTP to send GET and PUT requests to IGX.

The TCP Connection

When using HTTP to communicate with IGX, most HTTP libraries will handle the TCP connection automatically. However, if you're implementing your own HTTP stack, you'll need to set up the TCP socket and manage it yourself.

To establish a TCP socket at the target device IP address on **port 80**, you should follow these steps:

1. Open a TCP socket at the target device IP address on port 80.
2. Send an HTTP request message to the server.
3. Receive the HTTP response message from the server.

IGX can reuse one TCP connection for many requests (HTTP keep-alive), but only when the client asks for it. Send the header `Connection: keep-alive` on each request; the server answers with the

9. https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch
10. https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest/Using_XMLHttpRequest
11. https://curl.se/libcurl/
12. https://docs.libcpr.org/
13. https://github.com/elnormous/HTTPRequest
14. https://docs.microsoft.com/en-us/dotnet/api/system.net.http.httpclient?view=net-6.0
15. https://docs.microsoft.com/en-us/dotnet/api/system.net.httpwebrequest?view=net-6.0
16. https://openjdk.java.net/groups/net/httpclient/intro.html
17. https://hc.apache.org/httpcomponents-client-5.1.x/
18. https://www.ni.com/docs/en-US/bundle/labview/page/lvcomm/http_client_get.html
19. https://www.ni.com/docs/en-US/bundle/labview/page/lvcomm/http_client_put.html

same header and leaves the socket open. Without that header the server closes the connection after each response. Requests and response headers are plain ASCII text. Field files return a JSON body followed by a newline, and error responses have either an empty body or a short HTML message.

TCP Keep Alive

If you send requests frequently, reuse the connection. In a test against an FX4 on the local network, 50 sequential reads of `/io/heartbeat/value.json` averaged about 27 ms each over a kept-alive `requests.Session`, versus about 66 ms each when a new connection was opened for every request. The Python requests[20] library sends `Connection: keep-alive` by default and reuses the socket when you use a session object (see the examples later). Some HTTP/1.1 clients assume a persistent connection without sending the header; with IGX, their second request on the socket fails because the server has already closed it. Make sure your client sends the header, or opens a new connection for each request.

HTTP URLs

A URL (Uniform Resource Locator) is a standardized way of specifying the location of a resource on the internet. In the context of IGX, a URL is used to specify the path to an IO field. The URL consists of several parts that define the location and type of the resource being accessed.

The general format of a URL is:

```
scheme://host:port/path
```

- **Scheme:** Specifies the protocol being used to access the resource, such as HTTP or HTTPS.
- **Host:** Specifies the hostname or IP address of the device hosting the resource.
- **Port:** Specifies the port number on which the device is listening for requests. The default port for HTTP is 80 and typically does not need to be specified for most HTTP clients.
- **Path:** Specifies the location of the resource being accessed. In the context of IGX, the path would be in the format `/io/device/component/channel/value.json`.

For example, a URL to access the value of an IO field with the path `/io/device/component/channel/value` on a device with the IP address `192.168.1.100` would look like:

```
http://192.168.1.100/io/device/component/channel/value.json
```

By understanding the basics of URLs, developers can easily access and manipulate data generated by the control system using a variety of programming languages and tools that support HTTP communication.

HTTP Request Structure

To interact with IGX, programmers can send HTTP requests, which are plain ASCII text and have two main parts: the header and the body. The header and body are separated by new lines. With IGX, the only thing required in the header is the first line. HTTP uses standard "verbs" to differentiate the intentions of a request, and the verb is the first word in a request.

IGX supports three verbs: `GET` reads a file, `HEAD` returns only the headers a GET would return (useful for checking that a path exists), and `PUT` writes a file. Any other verb, including `POST` and `OPTIONS`, returns `501 Not Implemented`. Below is a minimal request. Each line ends with CR LF, and an empty line ends the header. Include `Connection: keep-alive` only if you will send more requests on the same socket:

```
GET /io/heartbeat/value.json HTTP/1.1
Host: 192.168.1.100
Connection: keep-alive
```

20. https://docs.python-requests.org/en/master/

In this example, the GET verb is used to request the value of an IO field with the path `/io/heartbeat/value`. The HTTP version used is 1.1. Take note in the use of two new line characters creating a blank line at the end of the message. This is the standard way to tell the sever that the message header or body is complete.

While it's useful to understand the structure of an HTTP request, most programming languages or frameworks come with HTTP support out of the box, so you typically won't have to worry about it. You'll only need to provide a URL to make a request.

For more extensive documentation on HTTP messages, you can refer to resources such as the Mozilla Developer Network (MDN) HTTP[21] Messages page.

HTTP Response Structure

When an HTTP request is sent to IGX, it responds with an HTTP response message. The response message also consists of a header and a body, which are separated by new lines. The header contains information about the response, such as the status code, while the body contains the response data.

The following is an example of a typical HTTP response for the `/io/heartbeat/value.json` URL from IGX:

```
HTTP/1.1 200 OK
Server: IGX
Access-Control-Allow-Origin: *
Access-Control-Allow-Credentials: true
Content-Length: 5
Content-Type: application/json
Content-Encoding: identity
Connection: keep-alive

true
```

In this example, the first line of the response message contains the HTTP version, status code, and status message. The status code of 200 indicates that the request was successful. The header also includes the Content-Type, which specifies the type of data in the response body. If you are requesting a JSON file, this type will always be `application/json`.

The heartbeat field is a Boolean, so the body is the JSON literal `true` followed by a newline, giving `Content-Length: 5`. The `Connection: keep-alive` header is present only because the request asked for it.

The body of a field file is exactly the JSON encoding of that field: a number, string, Boolean, array, or object, with no wrapper object around it. Parse the whole body as JSON.

Avoid requesting `/io/index.json` for the root node: it contains the entire device tree and is very large. Walk the tree with `children.json` instead.

URL	Contents
`/io/<path>/value.json`	The current value of an IO.
`/io/<path>/<field>.json`	Any other field of the node, such as `label`, `detail`, or `units`.
`/io/<path>/index.json`	The whole node as one JSON object: `name`, `label`, `detail`, `type`, `value_type`, `writable`, `permissions`, `value`, and all child nodes.

21. https://developer.mozilla.org/en-US/docs/Web/HTTP/Messages

URL	Contents
`/io/<path>/children.json`	The node's own fields plus the names of its children, each with an empty object. This is a cheap way to walk the tree one level at a time.
`/io/<path>/user_manual.md`	Generated user documentation for the node, in Markdown.

Besides one file per field, every node in `/io` provides generated files that describe it. Use these to find paths and check permissions without hard-coding them:

Discovering IO Paths and Metadata

IGX responds with a variety of HTTP status codes depending on the success or failure of the request. Some common HTTP status codes include:

- `200 OK`: a GET or HEAD succeeded, or a PUT body was delivered. A PUT reply has an empty body with `Content-Type: text/html`, and a 200 does not prove that the value changed (see Sending a PUT Request below).
- `400 Bad Request`: the URL could not be decoded or resolves outside the file system (for example, it contains `..`).
- `403 Forbidden`: the path is outside `/io/`, `/api/`, `/root/`, `/tmp/`, and `/ram/`. The body is a short HTML page.
- `404 Not Found`: a GET or HEAD for a path that does not exist. The body is empty.
- `500 Internal Server Error`: the path exists but could not be opened or read, or a PUT target could not be opened for writing. A PUT to a field name that does not exist under `/io` returns 500, not 404.
- `501 Not Implemented`: any verb other than GET, HEAD, or PUT.

Responses include `Access-Control-Allow-Origin: *`, so a web page served from another origin can read IO values with `fetch`. A cross-origin PUT is different: the browser first sends a CORS preflight `OPTIONS` request, IGX answers it with `501`, and the browser blocks the write. Pages served by the device itself are same-origin and can PUT normally. A page from another origin that needs to write should use the WebSocket protocol.

Browser Clients (CORS)

3.2.4 HTTP Python Examples

Python provides built-in support for HTTP communication, making it easy to interact with IGX. The requests library is commonly used for sending HTTP requests and handling responses. Here are some examples of how to use Python and the requests library to interact with IGX.

Python Requests Sessions

In the following examples we will be using the requests library. This library has two ways of making requests: directly from the `requests` module, or through a session object. Sessions are usually the better choice for production code because they keep the TCP connection alive between requests, but they make the examples slightly longer. A session keeps a separate connection pool for each host, so one session can serve several devices. Use one session per thread.

The following shows how to create a session object and use it to make requests.

```
import requests

session = requests.Session() # Create a session
session.get(...) # Make a GET request
session.put(...) # Make a PUT request
```

The following examples will use the form `requests.get` and `requests.put`, but if you are using a session, you can just replace the `requests` object with `session` and the example will still work.

Sending a GET Request

You can create a convenience function that queries an IP address with a given path, called a URL, and returns the translated JSON value. Here's an example of how to do this using the requests library:

```
import requests

def getURL(url):
  return requests.get(url, timeout=1.0).json()
```

Now you can create a more useful function that queries a device's hostname and returns the value:

```
def getHostName(ip):
  return getURL("http://" + ip + "/io/net/hostname/value.json")
```

The `getHostName` function takes an IP address as an argument and appends a fixed path to the file that contains the hostname value. You can then use this function to print out a local device hostname:

```
# Get the hostname of a device on our local network
print(getHostName("192.168.0.50"))
```

You can imagine creating a whole variety of convenient functions that consolidate these kinds of requests or combine multiple requests. This kind of encapsulation is highly encouraged, but keep in mind that URLs and IP addresses may change in future API versions, so your functions shouldn't be overly difficult to change in the future.

A more complete example:

```
import requests

# Target device IP address.
ip = "192.168.55.239"

# Helper function, does an HTTP GET and returns the parsed JSON value.
def getIOValue(path):
  return requests.get("http://" + ip + "/io" + path + "/value.json",
timeout=1.0).json()

# Helper function for a specific IO.
def getHostname():
  return getIOValue("/net/hostname")

def getHeartbeat():
  return getIOValue("/heartbeat")

# Requesting the device IO.
print(getHostname())
print(getHeartbeat())
```

In this example, the `getIOValue` function is used to send a GET request to IGX and return the parsed JSON value. The `getHostname` and `getHeartbeat` functions use `getIOValue` to request the hostname and heartbeat IO fields from IGX. Finally, the values of the hostname and heartbeat IO fields are printed to the console.

Sending a PUT Request

A PUT request writes a new value. It is structured like a GET request, except that the client supplies a body containing the JSON-encoded value, plus a `Content-Length` header. A `Content-Type` header is

optional because IGX always parses the body as JSON. Send a value of the field's type, and keep the double quotes around strings.

> ⚠ IGX answers `200 OK` with an empty body as soon as the body is delivered, even if the value does not change. A field that is not currently writable silently ignores the write. That includes a read-only IO, and a protected IO whose permission policy is locked. Check `writable` in the node's `index.json` before writing, and GET the value afterwards to confirm the write. Do not call `.json()` on a PUT response: the body is empty.

The following is an example of the new request structure:

```
PUT /io/net/hostname/value.json HTTP/1.1
Host: 192.168.1.100
Content-Length: 14

"New-Hostname"
```

In Python, sending a PUT request is easy using the `requests` library. Here's an example of how to use `requests` to send a PUT request to set the IO `/net/hostname` to `"New-Hostname"`:

```
import json
import requests

r = requests.put("http://<IP ADDRESS>/io/net/hostname/value.json", json.dumps("New-
Hostname"), timeout=1.0)
r.raise_for_status()
```

The `json.dumps` function is used to convert the value to a JSON-encoded string before sending it in the PUT request body.

Here's a more complete example:

```
import json
import requests

# Target device IP address.
ip = "192.168.55.239"

# One session reuses the TCP connection for every request.
session = requests.Session()

def ioURL(path, field="value"):
  return "http://" + ip + "/io" + path + "/" + field + ".json"

def getIOValue(path):
  r = session.get(ioURL(path), timeout=1.0)
  r.raise_for_status()
  return r.json()

def isIOWritable(path):
  r = session.get(ioURL(path, "index"), timeout=1.0)
  r.raise_for_status()
  return bool(r.json().get("writable", False))

def putIOValue(path, value):
  # IGX replies 200 with an empty body, so there is nothing to parse.
  r = session.put(ioURL(path), json.dumps(value), timeout=1.0)
  r.raise_for_status()
  # A 200 reply does not prove the write was applied, so read it back.
```

```
  return getIOValue(path) == value

# Helper functions for a specific IO.
def getHostname():
  return getIOValue("/net/hostname")

def putHostname(value):
  return putIOValue("/net/hostname", value)

print("Old Hostname", getHostname())
if isIOWritable("/net/hostname") and putHostname("New-Hostname"):
  print("New Hostname", getHostname())
else:
  print("Hostname was not changed (not writable right now)")
```

In this example, `putIOValue` sends the PUT, raises an exception on an HTTP error status, and then reads the value back to report whether the write took effect. `isIOWritable` reads the node's `index.json` to check permission before writing. `getHostname` and `putHostname` wrap these for the hostname IO. Comparing for exact equality works for strings, Booleans, and integers. For floating-point values, compare within a tolerance, because the device may round or clamp the value it stores.

3.2.5 Postman for HTTP Testing

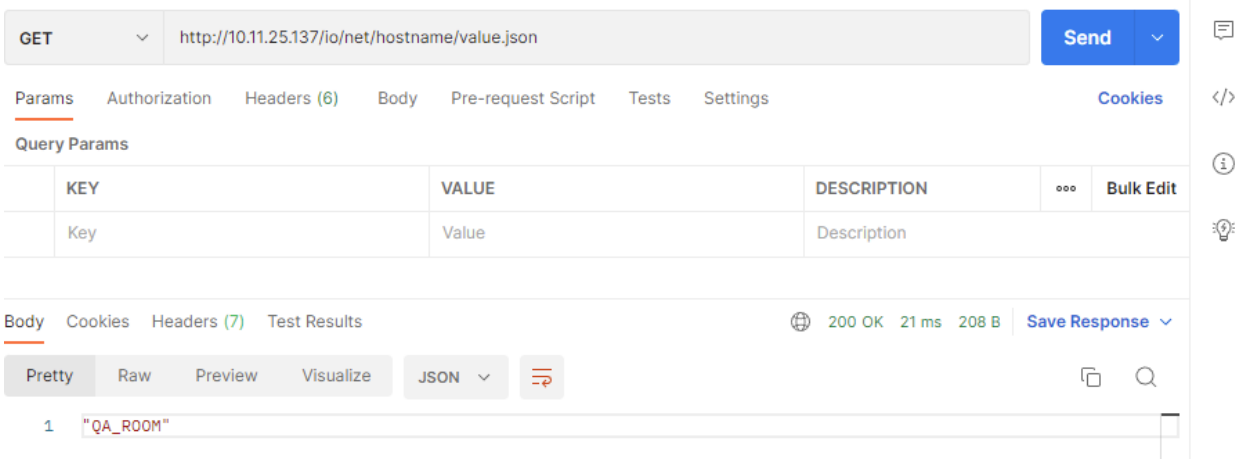
Postman is a powerful tool for sending and receiving HTTP messages with a user-friendly GUI interface. It can be invaluable for debugging your code and testing new queries without the hassle of programming. You can download Postman from the following link:

https://www.postman.com/downloads/

Sending a GET Request

To send a GET request using Postman, follow these steps:

1. Select GET method from the dropdown list.
2. Enter the target URL in the field at the top of the page.

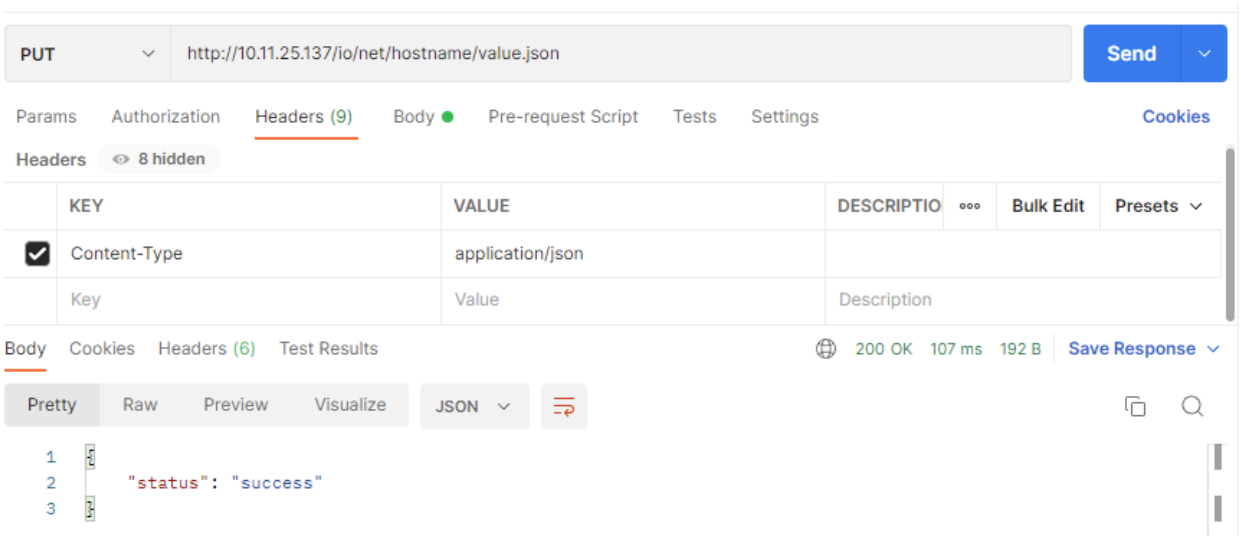


3. Click the Send button and observe the response.

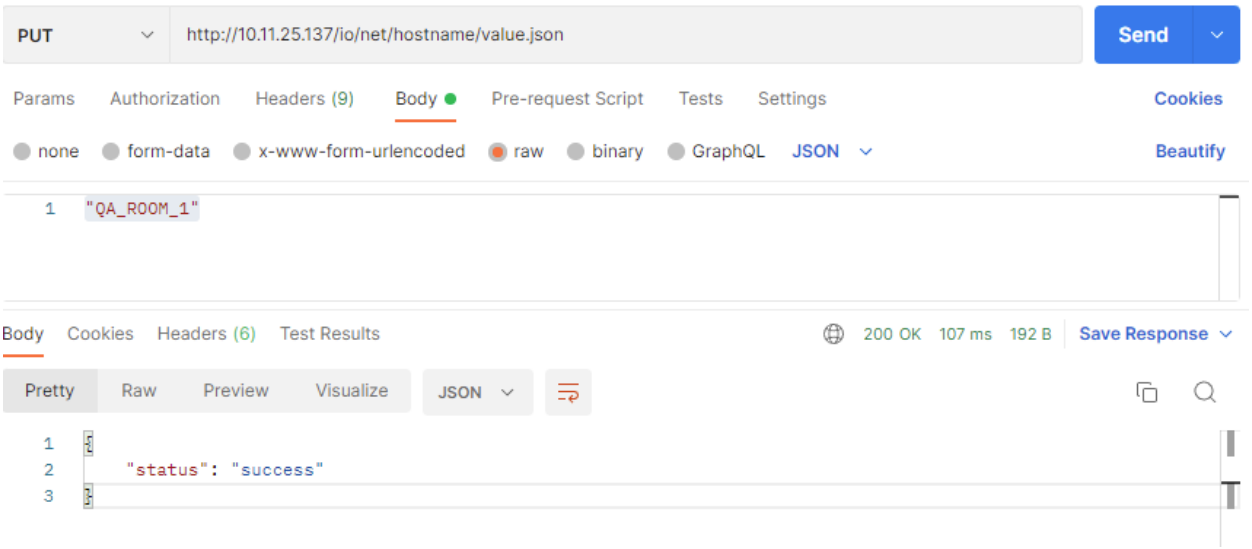
Sending a PUT Request

To send a PUT request using Postman, follow these steps:

1. Select PUT method.
2. Enter the target URL in the field at the top of the page.
3. Optionally, add the header `Content-Type: application/json`. IGX does not require it, because it always parses the body as JSON.



4. Add data to the Body field in **raw** format.



5. Click the Send button and observe the response.

3.2.6 PLC HTTP Programming

A Programmable Logic Controller (PLC) is a specialized computer that is used in industrial control systems to control various processes and machines. While most PLCs do not have built-in HTTP support, it is possible to make HTTP requests using a PLC in a number of ways.

One way to make HTTP requests from a PLC is to use a module or add-on that supports HTTP communication. Many PLC manufacturers offer modules or add-ons that can be added to a PLC to provide HTTP communication capabilities. These modules typically connect to the PLC via a standard interface such as Ethernet and may include a library or software that can be used to make HTTP requests.

The following is a list of some PLCs that support HTTP:

- Siemens S7-1500
- Allen-Bradley ControlLogix
- Mitsubishi iQ-R Series
- Beckhoff TwinCAT
- Wago PFC200
- Schneider Electric Modicon M580
- B&R Automation APROL

- Phoenix Contact ILC 2050 BI
- Omron NJ/NX series

Note that not all PLCs support HTTP out of the box and may require additional hardware or software modules to enable HTTP communication. Additionally, the implementation of HTTP support may vary between PLC manufacturers and models.

Converting Strings to Numeric or Binary Values

With HTTP, all data is transmitted as ASCII, human readable, strings, this makes the data very portable and easy to debug, however it requires the PLC to convert the strings back into their native types explicitly. This process can be prone to errors, so the following section includes some pointers and guidance we have found useful.

On Siemens PLCs, when using the STRG_VAL[22] instruction to convert from strings to numbers, there is a key parameter you should remember to use. Always use the `FORMAT` argument and use the `0002` value, this will use the exponential format with the "." character as the decimal point. This format aligns with the IGX formatting convention, where the “.” is consistently used to denote the decimal point, and an exponential suffix might be employed to reduce redundancy in zeros.

Protocol Gateways

Another way to make HTTP requests from a PLC is to use a gateway or protocol converter. These devices act as a bridge between the PLC and the network and can translate between different protocols such as Modbus and HTTP. By using a gateway or protocol converter, a PLC can be configured to make HTTP requests in the same way as any other network device. The following is a list of some possible gateway solutions:

- Anybus X-Gateway[23]
- Moxa NPort Gateway[24]
- Red Lion Data Station[25]

Note that Pyramid provides no guarantees for compatibility for these gateways. Please consult the specific gateway vendor and Pyramid together to ensure that your system will work.

Custom Code Solutions

It is also possible to write custom code for a PLC that implements HTTP communication. This can be done using a variety of programming languages and libraries, depending on the capabilities of the PLC and the requirements of the application. However, this approach requires a higher level of expertise and may not be suitable for all applications.

For example, some PLCs offer the capability to write custom function blocks or modules in languages like Structured Text, C/C++, or Java, which can then be used to perform HTTP communication. Additionally, some PLC programming environments provide built-in support for HTTP communication, such as the Siemens TIA Portal or the CODESYS Development System.

Writing custom code can offer more flexibility and control over the communication process, but it also requires more development effort and maintenance. It is important to carefully consider the requirements of the application and the capabilities of the PLC before deciding to write custom code.

The following is a simple example of custom C code for a Siemens S7-1500 PLC that implements the HTTP GET request:

```
#include <arpa/inet.h>
#include <netinet/in.h>
#include <sys/socket.h>
```

22. https://support.industry.siemens.com/cs/mdm/91696622?c=61295869963&lc=en-US
23. https://www.anybus.com/products/gateway-index/anybus-x-gateway
24. https://www.moxa.com/en/products/industrial-edge-connectivity/serial-device-servers/general-device-servers/nport-5100-series#overview
25. https://www.redlion.net/products/industrial-automation/controllers-and-data-acquisition/data-station-plus

```
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

// Define the URL of the resource being accessed
const char* url = "http://192.168.1.100/io/device/component/channel/value.json";

int main() {
    // Parse the URL to extract the host and path
    char host[128];
    char path[128];
    sscanf(url, "http://%127[^/]/%127s", host, path);

    // Define the request message using the extracted host and path
    char requestMessage[1024];
    snprintf(requestMessage, sizeof(requestMessage), "GET /%s HTTP/1.1\r\nHost:
%s\r\nConnection: close\r\n\r\n", path, host);

    // Open a TCP socket to the server
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    struct sockaddr_in serverAddr;
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_port = htons(80);
    serverAddr.sin_addr.s_addr = inet_addr("192.168.1.100");
    connect(sockfd, (struct sockaddr*)&serverAddr, sizeof(serverAddr));

    // Send the request message
    write(sockfd, requestMessage, strlen(requestMessage));

    // Read the response message
    char responseMessage[1024];
    read(sockfd, responseMessage, sizeof(responseMessage));

    // Close the TCP socket
    close(sockfd);

    // Parse the response message
    char* response = strstr(responseMessage, "\r\n\r\n") + 4;
    int value = atoi(response);

    // Do something with the value
    // ...

    return 0;
}
```

1. Include the necessary header files for sockets, string manipulation, and standard I/O operations.
2. Define the URL of the resource being accessed, which is a JSON file containing the value of a channel in an I/O device.
3. In the main function, parse the URL using `sscanf()` to extract the `host` and `path` components.
4. Construct the HTTP GET request message using the extracted `host` and `path` components with `snprintf()`.
5. Create a TCP socket using the `socket()` function and configure the server's address and port number.
6. Connect to the server using the `connect()` function.
7. Send the HTTP GET request message to the server using the `write()` function.

8. Read the server's response using the `read()` function and store it in a buffer (`responseMessage`).
9. Close the TCP socket using the `close()` function.
10. Parse the server's response to extract the value from the JSON file by finding the start of the actual response (after the HTTP header) using `strstr()`.
11. Convert the extracted value (as a string) to an integer using the `atoi()` function.
12. Perform any desired operation using the extracted value (value variable).

Note that this is just a simple example, and the actual implementation of HTTP communication will depend on the capabilities of the PLC and the requirements of the application. Additionally, custom code like this requires a higher level of expertise and may not be suitable for all applications.

3.3 IGX WebSocket Protocol Guide

3.3.1 WebSocket Overview

The WebSocket protocol enables two-way, real-time communication between a client and a server over a single, long-lived TCP connection. It starts as an HTTP request on the same port as the web server and is then upgraded, so it works with existing web infrastructure.

Every IGX device runs a WebSocket server on the same port as its HTTP server. It is the primary programmatic interface to the device: IGX's own web UI uses it for every value it displays and every command it sends. Through it a client can subscribe to any IO field or table, poll for changes (including every sample of a buffered signal), and write field values or press buttons.

This diagram depicts the sequence of events that take place when a client establishes a connection to a server using the TCP, HTTP, and WebSocket protocols.

The client initiates the connection by sending a TCP SYN message to the server, which responds with a SYN/ACK message. After the connection is established, the client sends an HTTP GET request to the server for a specific resource. The server responds with an HTTP 200 OK message, indicating that the request was successful.

At this point, the server sends an Upgrade header to the client, indicating that it supports the WebSocket protocol. The client then sends an Upgrade request to the server, which responds with an HTTP 101 Switching Protocols message. This indicates that the server has switched to the WebSocket protocol and the client can now send WebSocket data to the server. The sequence ends with both the client and server closing the WebSocket connection.

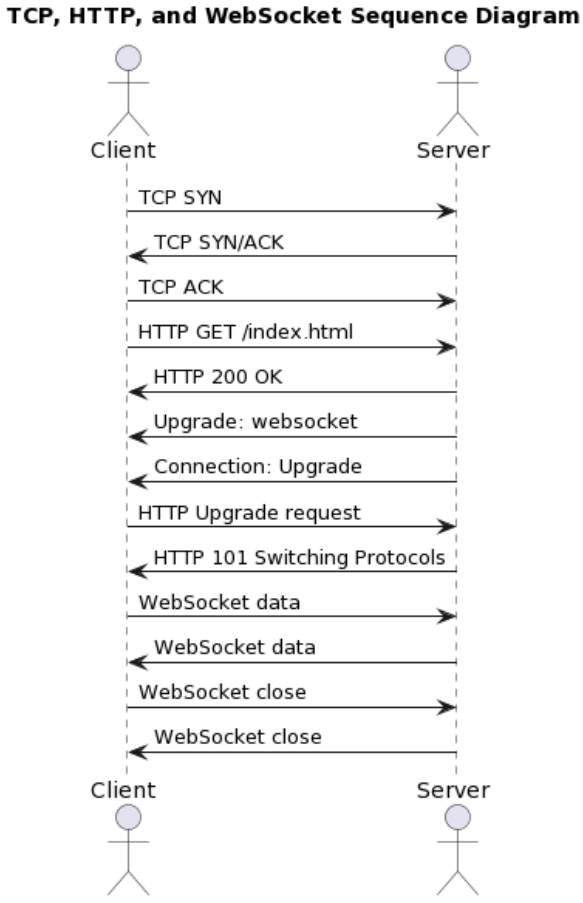


Figure 1 - WebSocket Timing Diagram

Connection Establishment: The client sends an HTTP `GET` with `Upgrade: websocket` and a `Sec-WebSocket-Protocol` header listing the data encodings it supports. The device answers `101 Switching Protocols`, naming the one encoding it selected, and the connection becomes a WebSocket.

Message Exchange: Every message in both directions is an object with an `event` name and an optional `data` payload. The client sends requests such as `subscribe`, `get`, and `set`; the device replies with `update`, `update_table`, and `update_id`.

Polling, not pushing: The device never sends field or table data on its own. It replies to each `get` with whatever changed since the previous `get`, so the client controls the update rate and can never be flooded.

Connection Termination: Either side may send a close frame. The device also drops connections that stay idle for 10 seconds (see Keep-alive and Idle Timeout).

Quick reference for client authors

- Connect to `ws://<device-ip>/` (port 80 by default) and always offer a subprotocol: `mpack.v2` for performance, `json.v1` for simplicity.
- Field paths start with `/` and end in `/value`, for example `/fx4/high_voltage/monitor_current/value`. Table paths are the bare table node, for example `/fx4/data_acquisition`.
- `subscribe` and `subscribe_table` **replace** the whole subscription list each time; they never add to it.
- Data arrives only in reply to `get`. Timestamps are integer **nanoseconds since the Unix epoch**.
- The device sends no error replies. Unknown paths, writes to read-only fields, and malformed messages are logged on the device and otherwise ignored, so confirm writes by watching the field.
- Send something (a `get` or a ping) at least every 10 seconds, or the device drops the connection.

3.3.2 Connecting

URL and Port

The WebSocket server listens on the same port as the HTTP server, configured by `/web_server/port/value` (default `80`). Connect to the root path, `ws://<device-ip>/` or `ws://<device-ip>:<port>/`. The device itself serves plain `ws://` only. To use `wss://` (TLS), put a reverse proxy in front of the device as described in the IGX Apache Proxy Server Guide (see page 46).

Subprotocol Negotiation

The client lists the encodings it supports in the `Sec-WebSocket-Protocol` request header. Each token is `<type>.<version>`, where type is `json` (UTF-8 JSON in text frames) or `mpack` (MessagePack in binary frames). The event names and message structure are identical in both encodings. The device answers with exactly one selected token:

Client offers	Device selects
Nothing	`json.v1`
One recognized token	That token, version suffix preserved
Several recognized tokens	The first token whose type matches the device's preferred protocol (`/web_server/preferred_protocol`, default `mpack`); otherwise MessagePack before JSON

Client offers	Device selects
Only unrecognized tokens	`json.v1`

Versions `v1` and `v2` are both accepted and echoed back unchanged; today the wire format is identical for both. Always offer at least one token. A device answering with a protocol the client did not offer is a handshake failure in browsers and in most libraries (Python's `websocket-client` raises `Invalid WebSocket Header`).

```
import websocket

ws = websocket.create_connection("ws://192.168.100.187/", subprotocols=["json.v1"])
print(ws.getheaders()["sec-websocket-protocol"])  # json.v1
```

To fall back gracefully, offer both (`subprotocols=["mpack.v2", "json.v1"]`) and check the selected header before encoding anything.

Keep-alive and Idle Timeout

A session that neither sends nor receives anything for `/web_server/session_manager/active_session_timeout` (default 10,000 ms) is considered dead and its TCP connection is dropped **without a close frame**, so the client sees an abrupt disconnect. A client that polls with `get` more often than that never hits the timeout. A client that only writes occasionally, or pauses polling, should send a WebSocket ping at least every few seconds; the device answers every ping with a pong.

The device never sends pings of its own. Do not send unsolicited pong frames: the device answers an incoming pong with a ping.

Close Codes

When the device closes a connection deliberately, the close frame carries one of these codes and a short ASCII reason:

Code	Reason	Meaning
1001	`process_exit`	IGX is shutting down.
1012	`igx_restart`, `firmware_update`, `os_reboot`	Planned restart. Reconnect after a short delay.
1008	`ip_filter`	The client's IP address was removed by the device's IP filter.
1011	`write_queue_overflow`	The client read too slowly: more than 4 MiB or 128 messages were queued for this connection.
1011	`web_memory_cap`	All web sessions together hit the memory cap (`/web_server/session_manager/memory_cap_bytes`, default 64 MiB). The newest WebSocket connection is closed first, so long-running clients keep streaming.
1011	`protocol_session_exception`	The device failed while handling one of the client's messages.

An abrupt close with no close frame (code 1006 in browsers) usually means the idle timeout expired or the device rebooted.

3.3.3 IGX Message Protocol

Every message is an object with an `event` string and, for most events, a `data` payload. In JSON sessions each message is one text frame; in MessagePack sessions each message is one binary frame containing a MessagePack map. Send each message as a single WebSocket frame (every common library does this by default).

```
{"event": "subscribe", "data": {"/fx4/high_voltage/monitor_current/value": true}}
```

Direction	Event	Purpose
Client to device	`config`	Set per-session options (short IDs, binary blobs, table options).
Client to device	`subscribe`	Replace the set of subscribed fields.
Client to device	`subscribe_table`	Replace the set of subscribed tables.
Client to device	`get`	Request an update for some or all subscriptions.
Client to device	`get_id`	Request the short-ID map for current subscriptions.
Client to device	`set`	Write one or more field values (including pressing buttons).
Device to client	`update`	Field data, in reply to get.
Device to client	`update_table`	Table config and rows, in reply to get.
Device to client	`update_id`	Short ID to path map, in reply to get_id.

Paths

Fields are addressed by their full IO path, starting with `/` and ending in `/value`, for example `/fx4/high_voltage/monitor_current/value`. Do not add the `/io` prefix used by the IGX HTTP Protocol Guide (see page 13). Paths that do not resolve to a field, including a field path missing its `/value` suffix, are silently ignored.

Tables are grids of rows and columns, such as a captured data acquisition buffer. They are addressed by their bare node path with no `/io` prefix and no `/value` suffix, for example `/fx4/data_acquisition`, and use their own `subscribe_table` and `update_table` events. `config`, `get`, `get_id`, `update_id`, and `set` apply to fields and tables alike.

To discover paths on a device, browse its web UI or read `/index.json` over HTTP for a subtree.

Values and Timestamps

Field data is always delivered as `[value, time]` pairs. `value` uses the field's natural type (number, string, boolean, or array). `time` is an unsigned 64-bit integer: **nanoseconds since 1970-01-01 UTC** on the device's clock, for example `1794603440735948045`.

> Nanosecond timestamps exceed 2^53, so JavaScript's `JSON.parse` rounds them to the nearest 256 ns. Use MessagePack with an int64-aware decoder, or treat sub-microsecond precision as unavailable in JSON. A device whose clock was never set reports times near the year 2000 (around `946684800000000000`).

Config Event

Optional. Sets per-session options. Unknown keys are ignored, and options can be changed at any time; they apply to existing subscriptions from the next update onward. Send it before `subscribe` so the first update already uses the options you want. All values must be booleans.

Example Message

```
{
  "event": "config",
  "data": {
    "use_short_id": true,
    "update_arrays_as_blobs": true,
    "update_buffered_as_blobs": true,
    "update_rows_as_blobs": true
  }
}
```

Configurable Parameters

Parameter	Default	Encodings	Description
`use_short_id`	false	Both	Key `update` and `update_table` data by an 8-character hex short ID instead of the full path. Send `get_id` to learn the mapping (see Update ID Event).
`always_update`	false	Both	Include every subscribed field in every update, even if unchanged. Useful as a heartbeat; can produce large messages.
`partial_row_update`	true	Both	Tables only. A large row commit is sometimes delivered in several chunks marked partial. When false, the device holds back a partial range until it completes.
`include_hidden_columns`	true	Both	Tables only. When false, hidden columns are left out of row data; cells then follow the visible columns only (those whose `column_hiddens` entry is false).
`update_arrays_as_blobs`	false	MessagePack	Encode float and double array field values as a binary blob instead of an array of numbers (see Binary Blobs).
`update_buffered_as_blobs`	false	MessagePack	Encode buffered float and double sample histories as one binary blob of values plus one of timestamps, instead of an array of pairs.
`update_rows_as_blobs`	false	MessagePack	Encode table rows as raw 32-bit cell words instead of per-cell arrays.

The JSON session only reads the first four keys; the three blob options are ignored over JSON.

JSON Schema

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Config Event",
  "type": "object",
  "properties": {
```

```
    "event": { "const": "config" },
    "data": {
      "type": "object",
      "properties": {
        "use_short_id": { "type": "boolean", "default": false },
        "always_update": { "type": "boolean", "default": false },
        "partial_row_update": { "type": "boolean", "default": true },
        "include_hidden_columns": { "type": "boolean", "default": true },
        "update_arrays_as_blobs": { "type": "boolean", "default": false },
        "update_buffered_as_blobs": { "type": "boolean", "default": false },
        "update_rows_as_blobs": { "type": "boolean", "default": false }
      }
    }
  },
  "required": ["event", "data"]
}
```

Subscribe Event

Sets which fields this session follows. `data` maps each field path to a boolean: `false` for **unbuffered** (only the latest value) or `true` for **buffered** (every sample since the previous `get`).

- **Replace, not add.** Each `subscribe` replaces the whole field list. To add a path, send the full list including the new path; to remove one, send the list without it; to clear everything, send `{}`.
- **Cheap to resend.** Resending an identical list is a no-op on the device. Fields already subscribed in the same mode keep their state; only new or changed entries are rebuilt.
- **First value.** The first `get` after a field is added returns its current value, so the client always gets a starting point.
- **Buffered window.** Each buffered subscription keeps at least the most recent 2048 samples (more if the field's own buffer is larger). If the client polls too slowly and more samples arrive than that, the oldest are dropped. Fields that do not keep a sample history silently fall back to unbuffered.
- **No data yet.** Subscribing does not send anything; send `get` to receive data.

Example Message

```
{
  "event": "subscribe",
  "data": {
    "/fx4/high_voltage/monitor_current/value": true,
    "/admin/version/value": false
  }
}
```

JSON Schema

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Subscribe Event",
  "type": "object",
  "properties": {
    "event": { "const": "subscribe" },
    "data": {
      "type": "object",
      "patternProperties": { "^/.+/value$": { "type": "boolean" } },
      "additionalProperties": false
    }
  },
  "required": ["event", "data"]
```

```
}
```

Subscribe Table Event

Sets which tables this session follows, keyed by bare table path. The boolean value is ignored; send `true`. Like `subscribe`, it replaces the whole table list, and `{}` clears it.

Unlike `subscribe`, every `subscribe_table` rebuilds **all** table subscriptions from scratch, so each listed table starts over with a full snapshot (complete config plus every stored row). Only resend it when the set of tables actually changes. Subscribing sends no data; follow it with `get`. A large snapshot can take several updates to arrive (see Update Table Event).

A data acquisition table's capture-state IO, `armed`, `active`, `sample_count`, and `buffer_size`, is not part of what `subscribe_table` / `update_table` reports. Those are ordinary sibling fields of the table, not entries in its config, so monitoring a capture in progress means subscribing to them separately with an ordinary `subscribe` event, for example `/fx4/data_acquisition/active/value`. See the IGX Data Acquisition Table Guide (see page 80) for a full worked example.

Example Message

```
{
  "event": "subscribe_table",
  "data": {
    "/fx4/data_acquisition": true
  }
}
```

JSON Schema

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Subscribe Table Event",
  "type": "object",
  "properties": {
    "event": { "const": "subscribe_table" },
    "data": {
      "type": "object",
      "patternProperties": { "^/.+$": { "type": "boolean" } },
      "additionalProperties": false
    }
  },
  "required": ["event", "data"]
}
```

Get Event

Asks the device for new data. Without `data`, or with an empty object, it covers all subscriptions. With selector keys it covers only the selected streams, which lets a client poll fast-moving buffered data and slow-moving status fields at different rates:

Selector	Covers
`"fields": true`	Unbuffered and buffered fields
`"fields_unbuffered": true`	Unbuffered fields only
`"fields_buffered": true`	Buffered fields only
`"tables": true`	Tables only

The replies to each kind of request are deterministic, which makes it possible to match replies to requests:

Request	Replies, in this order
`get` with no selectors	Exactly one `update` (possibly with empty data), then one `update_table` only if a subscribed table has something new
Any field selector	Exactly one `update`
`"tables": true`	Exactly one `update_table`, even if it is empty
Field and table selectors	Exactly one `update`, then exactly one `update_table`

The device processes messages from one connection strictly in order, so a client may keep a small number of `get` requests in flight (pipelining) to hide network latency. Keep the depth small (one or two per stream): every unread reply occupies the connection's outbound queue, and exceeding 4 MiB or 128 queued messages closes the connection with code 1011.

Example Messages

```
{"event": "get"}
{"event": "get", "data": {"fields_buffered": true}}
{"event": "get", "data": {"tables": true}}
```

JSON Schema

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Get Event",
  "type": "object",
  "properties": {
    "event": { "const": "get" },
    "data": {
      "type": ["object", "null"],
      "properties": {
        "fields": { "type": "boolean" },
        "fields_unbuffered": { "type": "boolean" },
        "fields_buffered": { "type": "boolean" },
        "tables": { "type": "boolean" }
      }
    }
  },
  "required": ["event"]
}
```

Set Event

Writes one or more fields. `data` maps field paths (the same form as `subscribe`) to new values. The device converts each value to the field's type.

Everything writable in IGX is a field, including buttons and editable table columns, so there is no separate command message. Setting a Button IO (see page 66) path to `true` presses it. See the IGX Data Acquisition Table Guide (see page 80) for a worked example with table buttons and columns.

- **No reply.** `set` produces no acknowledgement or error. Writes to unknown, read-only, or permission-protected fields are rejected and only logged on the device. Subscribe to the field and watch for the new value to confirm a write.

- **Several fields per message.** Fields in one message are written in the order the device parses them; send separate messages if order matters.
- **MessagePack integers.** Send integers as MessagePack integer types. Some encoders (for example JavaScript's) emit large whole numbers as float64, which integer fields may not accept.

Example Message

```
{
  "event": "set",
  "data": {
    "/fx4/high_voltage/current_limit/value": 1500.0,
    "/fx4/data_acquisition/collect_button/value": true
  }
}
```

JSON Schema

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Set Event",
  "type": "object",
  "properties": {
    "event": { "const": "set" },
    "data": {
      "type": "object",
      "patternProperties": {
        "^/.+$": { "type": ["string", "number", "boolean", "array", "object"] }
      },
      "additionalProperties": false
    }
  },
  "required": ["event", "data"]
}
```

Update Event

The device's reply to a `get` that covers fields. `data` maps each path (or short ID) to an array of `[value, time]` pairs. Only fields with something new are included, and an empty `data` object is a normal reply.

- **Unbuffered fields** always carry exactly one pair: the latest value.
- **Buffered fields** carry every sample since the previous `get`, oldest first. If nothing new arrived but the update was forced (first poll, or `always_update`), they carry one pair with the current value.
- In MessagePack sessions with `update_buffered_as_blobs`, buffered float and double fields arrive as a history blob instead of pairs, and with `update_arrays_as_blobs` array-valued fields arrive as array blobs (see Binary Blobs).

Example Message

```
{
  "event": "update",
  "data": {
    "/fx4/high_voltage/monitor_current/value": [
      [36.935287, 1794603440790948045],
      [60.286938, 1794603440805948045]
    ],
    "/admin/version/value": [["26.10.122034", 17946034401000000000]]
  }
}
```

JSON Schema

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Update Event",
  "type": "object",
  "properties": {
    "event": { "const": "update" },
    "data": {
      "type": "object",
      "additionalProperties": {
        "type": "array",
        "items": {
          "type": "array",
          "items": [
            { "type": ["string", "number", "boolean", "array", "object", "null"] },
            { "type": "integer", "minimum": 0, "description": "ns since Unix epoch" }
          ],
          "minItems": 2,
          "maxItems": 2
        }
      }
    }
  },
  "required": ["event", "data"]
}
```

Update Table Event

The device's reply to a `get` that covers tables. `data` maps each table path (or short ID) to an object. A table is included only if its config changed or it has rows to deliver, unless the request used the `tables` selector, in which case the event is always sent (possibly with empty `data`).

Key	Always present	Meaning
`oldest_row`	Yes	Physical row index of the oldest row. `0` for ordinary tables; for ring-buffer tables that have wrapped, the current write position. Read rows starting here to get chronological order.
`ring_capacity`	Yes	Ring size for wrap-aware clients; `0` means a linear table that never wraps.
`data_generation`	Yes	Data generation counter. It changes when the table is cleared or a new capture starts. When it changes, discard rows from the previous generation.
`config`	When changed	Table layout and metadata. Only keys that changed since the last update; the full set on the first update after subscribing.
`rows`	When rows are pending	New or changed rows (format below).

Table Config

Keys include `table_name`, `row_count`, `row_capacity`, `column_count`, `visible_column_count`, `column_names`, `column_types`, `column_units`, `column_formats`, `column_colors`, `column_hiddens`, `column_editables`, `column_store`, `column_channel_x`, `column_channel_y`, `column_channel_colors`,

`column_channel_visible_default`, `toolbar_action_paths`, `highlight_enabled`, `highlight_row_begin`, `highlight_row_end`, and `focus_row`. Tables may add others; ignore keys you do not recognize. `column_types` uses these codes: `0` Boolean, `1` signed 32-bit integer, `2` unsigned 32-bit integer, `3` 32-bit float.

A data acquisition table's capture state (`armed`, `active`, `sample_count`, `buffer_size`) is **not** included in this `config` object. Those IO are ordinary sibling fields of the table rather than entries in its config, so a client that needs to know when a capture starts or finishes should subscribe to them individually with a regular `subscribe` event (for example `/fx4/data_acquisition/active/value`) alongside its `subscribe_table` call. See the IGX Data Acquisition Table Guide (see page 80) for a full worked example.

Table Rows

By default `rows` is an array of `[row_index, [cell0, cell1, ...]]` entries. Cells follow the table's column order (all columns, or only the visible ones if `include_hidden_columns` is false), and `null` marks a cell that has no data yet. With `update_rows_as_blobs` (MessagePack only), each entry is instead `[first_row_index, <bin>]`, where the binary blob holds one or more consecutive rows (see Binary Blobs).

The device limits how many rows go into one update, adapting to how long serialization takes, so a large table can take many updates to deliver. Keep sending `get` (with `"tables": true`) until the rows you expect have arrived. Rows can also be redelivered when they change.

Example Message

```
{
  "event": "update_table",
  "data": {
    "/fx4/data_acquisition": {
      "oldest_row": 0,
      "ring_capacity": 0,
      "data_generation": 3,
      "config": {
        "table_name": "Data Acquisition",
        "row_count": 3,
        "row_capacity": 10000,
        "column_count": 2,
        "column_names": ["Time", "Value"],
        "column_types": [3, 3]
      },
      "rows": [
        [0, [0.0, 0.125]],
        [1, [0.001, 0.128]],
        [2, [0.002, null]]
      ]
    }
  }
}
```

JSON Schema

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Update Table Event",
  "type": "object",
  "properties": {
    "event": { "const": "update_table" },
    "data": {
```

```
      "type": "object",
      "additionalProperties": {
        "type": "object",
        "properties": {
          "oldest_row": { "type": "integer" },
          "ring_capacity": { "type": "integer" },
          "data_generation": { "type": "integer" },
          "config": { "type": "object" },
          "rows": {
            "type": "array",
            "items": {
              "type": "array",
              "items": [{ "type": "integer" }, { "type": "array" }],
              "minItems": 2,
              "maxItems": 2
            }
          }
        },
        "required": ["oldest_row", "ring_capacity", "data_generation"]
      }
    }
  },
  "required": ["event", "data"]
}
```

Update ID Event

The device's reply to `get_id`. When `use_short_id` is on, updates are keyed by short IDs (8 hex characters, for example `83A35538`) instead of full paths, which saves a lot of bandwidth on deep paths. `data` maps every current subscription, fields and tables alike, from short ID to full path.

The device sends `update_id` **only** in reply to `get_id`; it never announces new IDs on its own. After any `subscribe` or `subscribe_table` that adds paths, send `get_id` before relying on short keys. Keys that start with `/` are always full paths. Treat short IDs as opaque, and request the map again after every reconnect.

Example Message

```
{
  "event": "update_id",
  "data": {
    "83A35538": "/admin/version/value",
    "1F09C2D4": "/fx4/data_acquisition"
  }
}
```

JSON Schema

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Update ID Event",
  "type": "object",
  "properties": {
    "event": { "const": "update_id" },
    "data": {
      "type": "object",
      "patternProperties": { "^[0-9A-Fa-f]+$": { "type": "string", "pattern":
"^/" } },
      "additionalProperties": false
```

```
    }
  },
  "required": ["event", "data"]
}
```

Get ID Event

Asks the device for an `update_id` map of all current subscriptions. It needs no `data`.

Example Message

```
{"event": "get_id"}
```

JSON Schema

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Get ID Event",
  "type": "object",
  "properties": {
    "event": { "const": "get_id" },
    "data": {}
  },
  "required": ["event"]
}
```

Typical Protocol Sequence

1. Connect, offering `mpack.v2` and/or `json.v1`, and check which one the device selected.
2. Optionally send `config`.
3. Send `subscribe` and/or `subscribe_table`.
4. If `use_short_id` is on, send `get_id` and store the map from the `update_id` reply.
5. Send `get` (optionally with selectors) and process the `update` / `update_table` replies.
6. Repeat `get`, either right after each reply or on a timer, never letting more than 10 seconds pass without traffic.
7. On disconnect, reconnect and repeat from step 1. Subscriptions and short IDs do not survive a reconnect, and table subscriptions restart with a full snapshot.

The following is a basic example of how a typical sequence of events may happen:

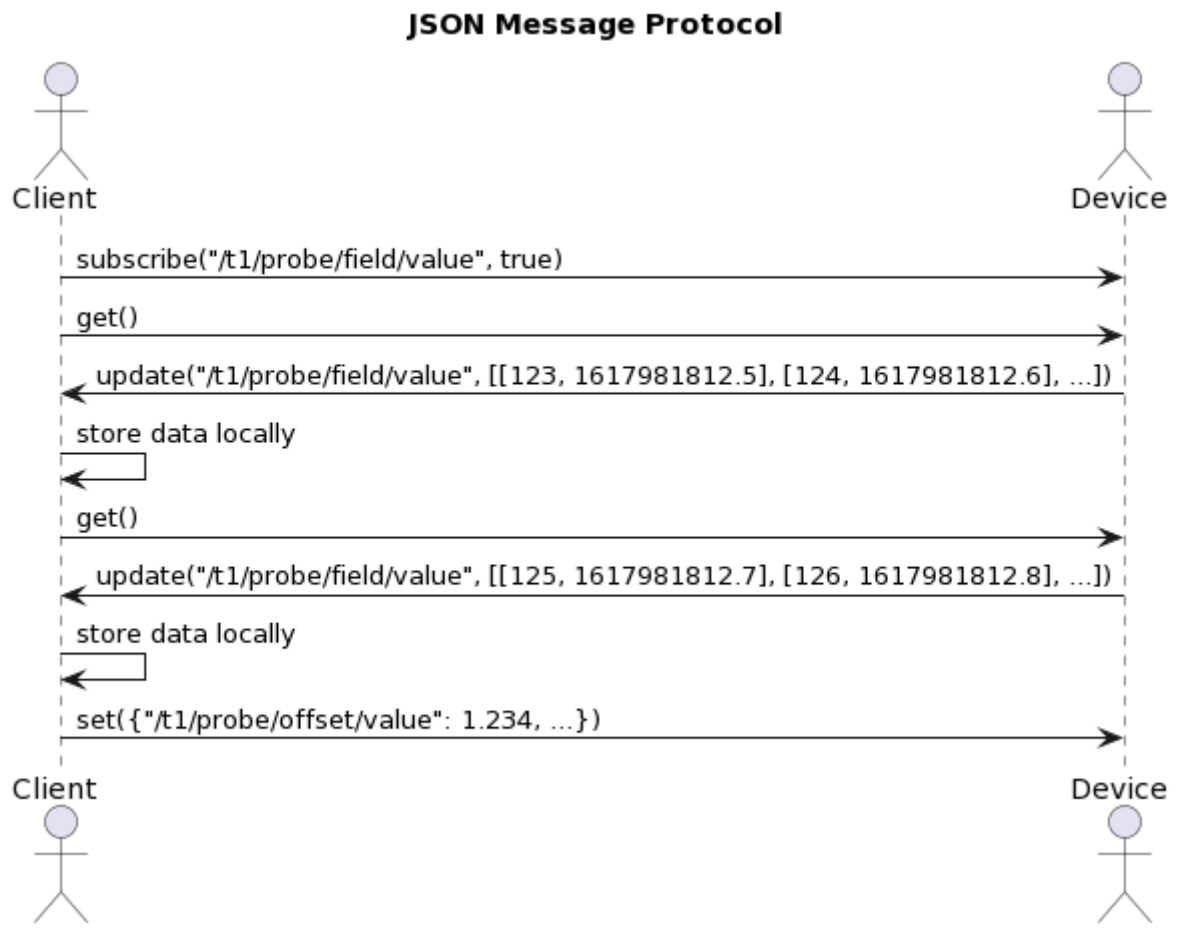


Figure 2 - JSON Event Timing Diagram

3.3.4 MessagePack Encoding and Binary Blobs

In an `mpack` session every message is a MessagePack map with the same `event` and `data` keys as JSON, sent as a binary frame. MessagePack is smaller and faster to encode and decode than JSON, and only MessagePack supports the binary blob options, which are by far the most efficient way to move numeric data.

⚠ Put the `event` key **before** the `data` key in every MessagePack map you send. The device reads the map in order and decides how to parse `data` from the event it has already seen. A message with `data` first is discarded. (Python dicts and JavaScript objects keep insertion order, so build them with `event` first.) Boolean values, such as subscription flags and config options, must be real MessagePack booleans, not 0 or 1.

All blobs are raw little-endian bytes copied from the device's memory. The dtype codes are:

$t	Element type
1	float32
2	float64

Array Blobs

With `update_arrays_as_blobs`, a float or double array value is sent as `{"$t": <dtype>, "$d": <bin>}` instead of an array of numbers. `$d` holds the elements back to back.

Buffered History Blobs

With `update_buffered_as_blobs`, the sample history of a buffered float or double field is sent as `{"$t": <dtype>, "$d": <bin>, "$ts": <bin>}` **in place of** the array of pairs. `$d` holds N values and `$ts` holds N uint64 timestamps (ns since the Unix epoch) in the same order. Forced single-value updates and non-numeric fields still use ordinary pairs, so decoders must handle both forms.

Table Row Blobs

With `update_rows_as_blobs`, each `rows` entry is `[first_row_index, <bin>]`. The blob holds R × C 32-bit little-endian cells, row-major, where C is the number of columns in the current layout (all columns, or only visible columns if `include_hidden_columns` is false) and R is the blob size divided by 4C. The rows are `first_row_index` to `first_row_index + R - 1`. Interpret each column by its `column_types` code: `3` float32, `1` int32, `2` uint32, `0` Boolean (low byte non-zero). The word `0x7FFFFFFF` marks an empty cell; for float columns that bit pattern is a NaN, so empty float cells decode as NaN automatically.

```
import numpy as np

def decode_field_value(v):
    if isinstance(v, dict) and "$ts" in v:            # buffered history blob
        dtype = np.float32 if v["$t"] == 1 else np.float64
        return np.frombuffer(v["$d"], dtype=dtype), np.frombuffer(v["$ts"],
dtype="<u8")
    if isinstance(v, dict) and "$d" in v:             # array blob
        return np.frombuffer(v["$d"], dtype=np.float32 if v["$t"] == 1 else
 np.float64)
    return v                                          # ordinary [[value, time], ...]

def decode_row_blob(first_row, blob, ncols):
    cells = np.frombuffer(blob, dtype="<u4").reshape(-1, ncols)
    return range(first_row, first_row + len(cells)), cells   # cells.view("<f4") for
float columns
```

3.3.5 Performance Guidance

These are the practices IGX's own web UI uses to stream hundreds of columns at kilohertz rates. Apply them in order of impact:

1. **Use MessagePack with blobs.** Offer `mpack.v2` and enable `update_buffered_as_blobs`, `update_arrays_as_blobs`, and `update_rows_as_blobs`. Numeric data then arrives as contiguous bytes that can be wrapped without copying (for example `numpy.frombuffer` or a JavaScript `Float32Array`).
2. **Turn on short IDs** (`use_short_id`) and resolve them from one `get_id` after each subscription change.
3. **Use buffered subscriptions for signals and unbuffered for status.** Buffered mode delivers every sample regardless of poll rate, so a slower poll yields bigger, more efficient batches without losing data, as long as each poll interval covers fewer than 2048 samples.
4. **Poll each stream separately.** Use `get` selectors to poll buffered fields, unbuffered fields, and tables at independent rates (the web UI uses 16–32 ms for active buffered and table streams and 100–300 ms for status fields).
5. **Pipeline one or two gets per stream.** Send the next `get` before the previous reply arrives, so network round-trip time does not cap throughput. Match replies to requests in order (see Get Event).
6. **Disable Nagle's algorithm** (TCP_NODELAY) on the client socket so small requests are sent immediately.

7. **Keep subscriptions stable.** Do not resend `subscribe_table` unless the table set changes (it restarts every table snapshot), and avoid `always_update`.
8. **Use one connection per client.** Each connection costs device memory and CPU, and the device sheds the newest connections first under memory pressure.

Poll rate is a trade-off between latency and device CPU. Measured on an FX4 simulator, a single buffered float field at about 40 Hz delivered every sample both ways: a 20 ms poll took about 380 messages per 10 seconds (one or two samples each), while a 50 ms poll took about 190 messages for the same data. Poll fast only when latency matters.

3.3.6 Suitable WebSocket Libraries

Any RFC 6455 client library works. This guide uses Python; the others work the same way. For MessagePack sessions, pair the WebSocket library with a MessagePack library that supports binary types and 64-bit integers.

Language	WebSocket library	MessagePack library
Python	websocket-client[26] or websockets[27] (asyncio)	msgpack[28]
JavaScript (browser)	WebSocket API[29] (set `binaryType = "arraybuffer"`)	msgpackr[30] (used by the IGX web UI)
JavaScript (Node.js)	ws[31] or the built-in WebSocket (Node 22+)	msgpackr[32]
C++	Boost.Beast[33] or WebSocket++[34]	msgpack-c[35]
C	libwebsockets[36]	mpack[37]
C#	ClientWebSocket[38] (built in)	MessagePack-CSharp[39]
Java	Java-WebSocket[40]	msgpack-java[41]
Go	gorilla/websocket[42]	vmihailenco/msgpack[43]
Rust	tungstenite[44]	rmp-serde[45]
Swift	Starscream[46]	
LabVIEW	LabVIEW WebSockets API[47] (unofficial)	Use JSON

26. https://github.com/websocket-client/websocket-client
27. https://github.com/python-websockets/websockets
28. https://github.com/msgpack/msgpack-python
29. https://developer.mozilla.org/en-US/docs/Web/API/WebSocket
30. https://github.com/kriszyp/msgpackr
31. https://github.com/websockets/ws
32. https://github.com/kriszyp/msgpackr
33. https://www.boost.org/doc/libs/release/libs/beast/
34. https://github.com/zaphoyd/websocketpp
35. https://github.com/msgpack/msgpack-c
36. https://github.com/warmcat/libwebsockets
37. https://github.com/ludocode/mpack
38. https://learn.microsoft.com/dotnet/api/system.net.websockets.clientwebsocket
39. https://github.com/MessagePack-CSharp/MessagePack-CSharp
40. https://github.com/TooTallNate/Java-WebSocket
41. https://github.com/msgpack/msgpack-java
42. https://github.com/gorilla/websocket
43. https://github.com/vmihailenco/msgpack
44. https://github.com/snapview/tungstenite-rs
45. https://github.com/3Hren/msgpack-rust
46. https://github.com/daltoniam/Starscream
47. https://www.ni.com/en-us/support/downloads/tools-network/download.websockets-api.html#374385

3.3.7 Python T1 Example

This example uses JSON to collect magnetic field samples from a T1 device for a fixed time and write them to a CSV file. It is the simplest correct client: one buffered subscription and one get in flight at a time.

Prerequisites: Python 3.8+ and `pip install websocket-client`.

```
import csv
import json
import time

import websocket

IP = "192.168.55.239"            # device IP address
COLLECTION_TIME = 2.0            # seconds to collect
OUTPUT_FILE = "t1_data.csv"
FIELD = "/t1/probe/field/value"

ws = websocket.create_connection(f"ws://{IP}/", subprotocols=["json.v1"], timeout=5)


def send_event(event, data=None):
    message = {"event": event}
    if data is not None:
        message["data"] = data
    ws.send(json.dumps(message))


# true = buffered: every sample since the previous get, not just the latest one
send_event("subscribe", {FIELD: True})
send_event("get")

samples = []
start = time.time()
while time.time() - start < COLLECTION_TIME:
    message = json.loads(ws.recv())
    if message["event"] != "update":
        continue
    samples += message["data"].get(FIELD, [])
    time.sleep(0.05)             # let a batch of samples accumulate
    send_event("get")            # exactly one get in flight at a time

ws.close()

with open(OUTPUT_FILE, "w", newline="") as file:
    writer = csv.writer(file)
    writer.writerow(["value", "time_ns", "time_s"])
    for value, time_ns in samples:
        writer.writerow([value, time_ns, time_ns / 1e9])

print("Collected", len(samples), "samples, written to", OUTPUT_FILE)
```

Each reply carries every sample recorded since the previous `get`, so the 50 ms sleep does not lose data; it only batches it. The loop also keeps the connection well inside the 10-second idle timeout.

3.3.8 High-Performance Python Client

This client applies every item from Performance Guidance: MessagePack with all three blob options, short IDs, separate pipelined poll loops for buffered fields, unbuffered fields, and tables,

TCP_NODELAY, and zero-copy NumPy decoding. Use it as a template for production clients, or port it to another language.

Prerequisites: Python 3.8+ and `pip install websocket-client msgpack numpy`.

```
"""High-throughput IGX WebSocket client (MessagePack + binary blobs).

pip install websocket-client msgpack numpy
"""
import socket
import time

import msgpack
import numpy as np
import websocket

DTYPES = {1: np.float32, 2: np.float64}
CELL_DTYPES = {0: np.uint32, 1: np.int32, 2: np.uint32, 3: np.float32}
EMPTY_CELL = 0x7FFFFFFF


class IGXFastClient:
    def __init__(self, host, port=80, pipeline_depth=2, min_interval=0.05):
        self.url = f"ws://{host}:{port}/"
        self.pipeline_depth = pipeline_depth  # gets kept in flight per stream
        self.min_interval = min_interval      # seconds between gets per stream
        self.fields = {}                      # path -> buffered flag
        self.tables = {}                      # path -> True
        self.short_to_path = {}
        self.table_layout = {}                # path -> list of column type codes
        self.awaiting_ids = False
        self.ws = None

    # ---- connection ----------------------------------------------------

    def connect(self):
        self.ws = websocket.create_connection(
            self.url,
            subprotocols=["mpack.v2"],
            sockopt=((socket.IPPROTO_TCP, socket.TCP_NODELAY, 1),),
            skip_utf8_validation=True,
            timeout=5,
        )
        selected = self.ws.getheaders().get("sec-websocket-protocol", "")
        if not selected.startswith("mpack"):
            raise RuntimeError(f"device selected {selected!r}, expected mpack")
        self.send("config", {
            "use_short_id": True,
            "update_arrays_as_blobs": True,
            "update_buffered_as_blobs": True,
            "update_rows_as_blobs": True,
            "include_hidden_columns": False,
        })
        self.send("subscribe", self.fields)
        self.send("subscribe_table", self.tables)
        self.send("get_id")
        self.awaiting_ids = True

    def close(self):
```

```
        if self.ws:
            self.ws.close()
            self.ws = None

    def send(self, event, data=None):
        msg = {"event": event} if data is None else {"event": event, "data": data}
        self.ws.send_binary(msgpack.packb(msg))

    def recv(self):
        _, payload = self.ws.recv_data()
        return msgpack.unpackb(payload, raw=False)

    # ---- decoding ------------------------------------------------------

    def resolve(self, key):
        return key if key.startswith("/") else self.short_to_path.get(key)

    @staticmethod
    def decode_field(value):
        """Returns (values ndarray, times_ns ndarray) for any field update."""
        if isinstance(value, dict) and "$ts" in value:
            vals = np.frombuffer(value["$d"], dtype=DTYPES[value["$t"]])
            times = np.frombuffer(value["$ts"], dtype="<u8")
            return vals, times
        vals = [v for v, _ in value]
        times = np.array([t for _, t in value], dtype=np.uint64)
        return vals, times

    def decode_rows(self, path, rows):
        """Yields (first_row_index, cells) per row block.

        cells is a (n_rows, n_columns) uint32 matrix of raw PackedIO words.
        Use column() to view one column with its real type.
        """
        ncols = len(self.table_layout.get(path, []))
        for start, cells in rows:
            if isinstance(cells, (bytes, bytearray)) and ncols:
                yield start, np.frombuffer(cells, dtype="<u4").reshape(-1, ncols)
            else:
                # Per-cell form ([row, [cell, ...]]) when blobs are off.
                yield start, np.array([cells], dtype=object)

    def column(self, path, block, c):
        """One column of a decoded row block. Empty cells: NaN (float) / masked (int,
bool)."""
        t = self.table_layout[path][c]
        raw = block[:, c]
        if t == 3:
            return raw.view("<f4")  # empty sentinel 0x7FFFFFFF is a NaN bit pattern
        empty = raw == EMPTY_CELL
        if t == 0:
            return np.ma.masked_array((raw & 0xFF) != 0, empty)
        return np.ma.masked_array(raw.view(CELL_DTYPES[t]), empty)

    # ---- streaming loop ------------------------------------------------

    def run(self, on_field, on_rows=None, duration=None):
        self.connect()
```

```
        streams = []
        if any(self.fields.values()):
            streams.append(("buffered", {"fields_buffered": True}, "update"))
        if not all(self.fields.values()):
            streams.append(("unbuffered", {"fields_unbuffered": True}, "update"))
        if self.tables:
            streams.append(("tables", {"tables": True}, "update_table"))

        in_flight = {name: 0 for name, _, _ in streams}
        last_sent = {name: 0.0 for name, _, _ in streams}
        pending = []  # FIFO of stream names; the device answers gets in order
        deadline = None if duration is None else time.monotonic() + duration

        try:
            while deadline is None or time.monotonic() < deadline:
                now = time.monotonic()
                next_due = None
                for name, selector, _ in streams:
                    if in_flight[name] >= self.pipeline_depth:
                        continue
                    due = last_sent[name] + self.min_interval
                    if now >= due:
                        self.send("get", selector)
                        in_flight[name] += 1
                        last_sent[name] = now
                        pending.append(name)
                    elif next_due is None or due < next_due:
                        next_due = due

                # Nothing outstanding: wait for the next get instead of blocking on
recv().
                if not pending and not self.awaiting_ids:
                    if next_due is not None:
                        time.sleep(max(0.0, next_due - time.monotonic()))
                    continue

                msg = self.recv()
                event, data = msg.get("event"), msg.get("data") or {}

                if event == "update_id":
                    self.short_to_path.update(data)
                    self.awaiting_ids = False
                    continue

                if event in ("update", "update_table") and pending:
                    in_flight[pending.pop(0)] -= 1

                if event == "update":
                    for key, value in data.items():
                        path = self.resolve(key)
                        if path:
                            on_field(path, *self.decode_field(value))
                elif event == "update_table":
                    for key, table in data.items():
                        path = self.resolve(key)
                        if not path:
                            continue
                        cfg = table.get("config", {})
```

```
                        if "column_types" in cfg or "column_hiddens" in cfg:
                            types = cfg.get("column_types", [])
                            hidden = cfg.get("column_hiddens", [False] * len(types))
                            self.table_layout[path] = [t for t, h in zip(types,
hidden) if not h]
                        if on_rows and "rows" in table:
                            on_rows(path, table, self.decode_rows(path,
table["rows"]))
        finally:
            self.close()
```

Usage:

```
client = IGXFastClient("192.168.100.187", min_interval=0.05)
client.fields = {
    "/fx4/high_voltage/monitor_current/value": True,    # buffered signal
    "/admin/version/value": False,                      # unbuffered status
}
client.tables = {"/fx4/data_acquisition": True}

def on_field(path, values, times_ns):
    # values: ndarray (or list for non-numeric fields); times_ns: uint64 ndarray
    print(path, len(values), "samples, last", values[-1])

def on_rows(path, table, blocks):
    for first_row, cells in blocks:
        first_column = client.column(path, cells, 0)
        print(path, "rows", first_row, "to", first_row + len(cells) - 1,
first_column[:3])

client.run(on_field, on_rows, duration=10.0)
```

Notes on the design:

- Every `get` uses an explicit selector, so each produces exactly one reply of a known type, and a simple first-in, first-out list matches replies to streams.
- `pipeline_depth` gets per stream are kept in flight, spaced at least `min_interval` apart. When nothing is in flight the loop sleeps instead of blocking on a reply that will never come.
- `decode_field` returns NumPy views straight over the received bytes for blob updates, and handles plain pairs for forced single-value or non-numeric updates.
- Table layouts are tracked from `config` so row blobs can be reshaped; hidden columns are excluded via `include_hidden_columns`. Check `data_generation` in `on_rows` if your table can be cleared or recaptured.
- For unattended use, wrap run() in a reconnect loop with a short back-off. The client sets up its subscriptions again on every connect.

Verified against an FX4 simulator: over 10 seconds the client received every sample of a buffered field (no duplicates or out-of-order timestamps) in about 190 messages at the default 50 ms interval.

3.3.9 Troubleshooting

Symptom	Likely cause
Handshake fails with `Invalid WebSocket Header` or a browser connection error	No recognized subprotocol was offered, so the device answered with json.v1, which the client did not offer.
Connection drops after about 10 seconds	No traffic within the idle timeout. Poll or send pings.

Symptom	Likely cause
A subscribed path never appears in `update`	The path is wrong (missing `/value`, or an `/io` prefix), or the field has not changed. Unchanged fields are omitted except on the first poll.
Updates arrive keyed by codes like `83A35538`	`use_short_id` is on. Send `get_id` and map the keys.
A `set` has no effect	The field is read-only, permission-protected, or the path is wrong. Check the device log.
Buffered data has gaps	More than 2048 samples arrived between polls. Poll more often.
Table rows restart from row 0 repeatedly	`subscribe_table` is being resent; each resend restarts the snapshot.
MessagePack messages are ignored	`data` was placed before `event` in the map, or booleans were sent as integers.
Closed with code 1011	The client is reading too slowly (write_queue_overflow) or the device hit its web memory cap (web_memory_cap).

3.3.10 Conclusion

The IGX WebSocket protocol gives a client complete, efficient access to a device: subscribe to fields and tables, poll for changes at a rate you choose, and write values with `set`. Start from the Python T1 Example for simple scripts, and from the High-Performance Python Client for production or high-rate data. For table capture workflows, see the IGX Data Acquisition Table Guide (see page 80).

3.4 IGX Apache Proxy Server Guide

This guide explains how to configure an Apache proxy server to dynamically route requests to different backend devices based on paths and ensure compatibility with WebSocket connections, upgrading all traffic to HTTPS and WSS.

3.4.1 Prerequisites

1. Apache installed and running.
2. Ensure you have administrative access to edit Apache configuration files.
3. Enable required Apache modules (explained below).

3.4.2 Setup Notes

1. The proxy server upgrades all traffic to HTTPS and WSS.
2. Ensure backend devices are accessible from the proxy server.
3. Verify that the paths and IP addresses match your network configuration.

3.4.3 Steps to Configure Apache Proxy

1. Edit the Apache Configuration File

Locate and edit the Apache virtual host configuration file (e.g., `/etc/apache2/sites-available/000-default.conf`). Add the following directives:

```
<VirtualHost *:443>
    ServerName example.com
```

```
    ProxyRequests Off
    ProxyPreserveHost On

    SSLEngine On
    SSLCertificateFile /path/to/your/certificate.crt
    SSLCertificateKeyFile /path/to/your/private.key

    <Proxy *>
        Require all granted
    </Proxy>

    # Proxy settings for specific devices
    ProxyPass "/device1/" "<http://192.168.0.1/>"
    ProxyPassReverse "/device1/" "<http://192.168.0.1/>"

    ProxyPass "/device2/" "<http://192.168.0.2/>"
    ProxyPassReverse "/device2/" "<http://192.168.0.2/>"

    # Catch-all for other devices (optional)
    ProxyPass "/device/" "<http://192.168.0.x/>"
    ProxyPassReverse "/device/" "<http://192.168.0.x/>"

    # Dynamic mapping for WebSocket
    RewriteEngine On
    RewriteCond %{HTTP:Upgrade} websocket [NC]
    RewriteCond %{HTTP:Connection} upgrade [NC]
    RewriteRule ^/device1/(.*) ws://192.168.0.1/$1 [P,L]
    RewriteRule ^/device2/(.*) ws://192.168.0.2/$1 [P,L]

    # Example for dynamic routing based on device number
    RewriteRule "^/device(\d+)/(.*)" "ws://192.168.0.%1/$2" [P,L]
</VirtualHost>

<VirtualHost *:80>
    ServerName example.com
    RewriteEngine On
    RewriteRule ^(.*)$ <https://%{HTTP_HOST}$1> [R=301,L]
</VirtualHost>
```

2. Enable Required Apache Modules

Run the following commands to enable the necessary modules for proxying and WebSocket support:

```
sudo a2enmod proxy
sudo a2enmod proxy_http
sudo a2enmod proxy_wstunnel
sudo a2enmod rewrite
sudo a2enmod ssl
sudo systemctl restart apache2
```

3. Verify Configuration

Ensure the configuration routes both HTTP and WebSocket traffic correctly. Test the setup using tools like `curl` and `wscat`.

Example Commands

Testing HTTP Routing:

```
curl https://example.com/device1/io/net/hostname/value.json
```

This should return the device's hostname in a JSON string.

Testing WebSocket Routing:
Install `wscat` if not already installed:

```
npm install -g wscat
```

Connect to a WebSocket endpoint:

```
wscat -c wss://example.com/device1/
```

3.4.4 Explanation of Configuration

ProxyPass and ProxyPassReverse

- `ProxyPass` forwards incoming requests to the specified backend server.
- `ProxyPassReverse` ensures the response headers are correctly rewritten for the client.

Dynamic Mapping with RewriteRule

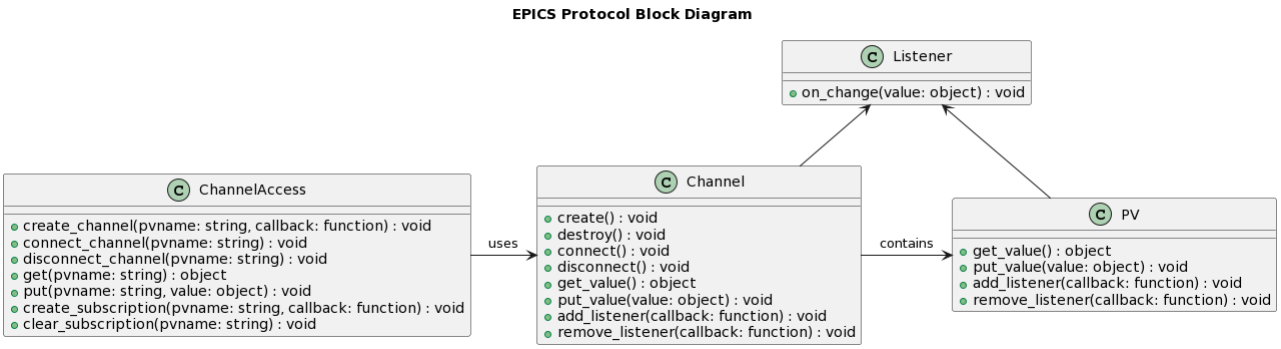
- The `RewriteRule` directive dynamically maps URLs to backend devices.
- Example: `/device1/path` maps to `<http://192.168.0.1/path>`.
- WebSocket-specific rules handle protocol upgrades with `Upgrade` and `Connection` headers.

3.5 IGX EPICS Protocol Guide

3.5.1 EPICS Overview

The Experimental Physics and Industrial Control System (EPICS) is a widely used, open-source, distributed control system for scientific instruments, such as particle accelerators, telescopes, and large-scale experiments. EPICS facilitates communication between hardware devices and software applications, enabling efficient data acquisition, device control, and monitoring.

The core of EPICS is the Channel Access protocol, which allows efficient and scalable communication between servers (Input/Output Controllers, or IOCs) and clients (user interfaces, scripts, or other applications). IOCs are responsible for interfacing with hardware devices and exposing their data as Process Variables (PVs). Clients can read or write to these PVs through the Channel Access protocol, which supports both synchronous and asynchronous communication.



EPICS has a modular architecture and supports various device types and communication interfaces through its extensive library of device drivers and protocol modules. This flexibility allows users to integrate a wide range of hardware devices and create custom control systems tailored to their specific needs.

For more information on EPICS, you can refer to the following resources:

1. EPICS Website: https://epics-controls.org/
2. EPICS Wiki: https://epics.anl.gov/

These resources provide in-depth information about EPICS, its architecture, components, and applications. They also include guidelines for getting started, tutorials, and examples to help users effectively utilize EPICS in their projects.

EPICS Network Protocols and Ports

When a device configuration includes an `epics_server` node, IGX runs a Channel Access server and sets `EPICS_CAS_SERVER_PORT` to 5064 and `EPICS_CAS_BEACON_PORT` to 5065. A device without that node does not open these ports and does not answer PV searches.

TCP 5064 carries reads, writes, and subscriptions. UDP 5064 carries PV name searches: a client broadcasts a PV name, and the device replies if it hosts that PV. UDP 5065 carries beacons, which announce the server so clients can find it. Client-side Channel Access repeaters also use UDP 5065. IGX does not listen on TCP 5065.

Protocol	Port	Description
TCP	5064	Channel Access server. Clients connect here for reads, writes, and monitors.
UDP	5064	PV name search. Clients broadcast a PV name; the device replies if it hosts that PV.
UDP	5065	Beacons. The device announces itself so clients can find it. Client-side CA repeaters use this port too.

A firewall between the client and the device has to allow UDP and TCP 5064, and UDP 5065. Name search is a UDP broadcast, so it stays on the local subnet unless the client is pointed at the device with `EPICS_CA_ADDR_LIST` (see below).

The timing diagram shows the sequence of events that occurs during a typical exchange between a client and server using the EPICS network protocol.

The client initiates the exchange by sending a Channel Access request to the server. The server responds with a Channel Access response, and the client then sends a series of requests to create and connect to the channel, get and put values, and create and clear subscriptions.

Whenever a process variable's value changes, the server sends a value change notification to the client, which then processes the change.

Finally, the client sends a request to disconnect from the channel, and the server responds with a disconnect response.

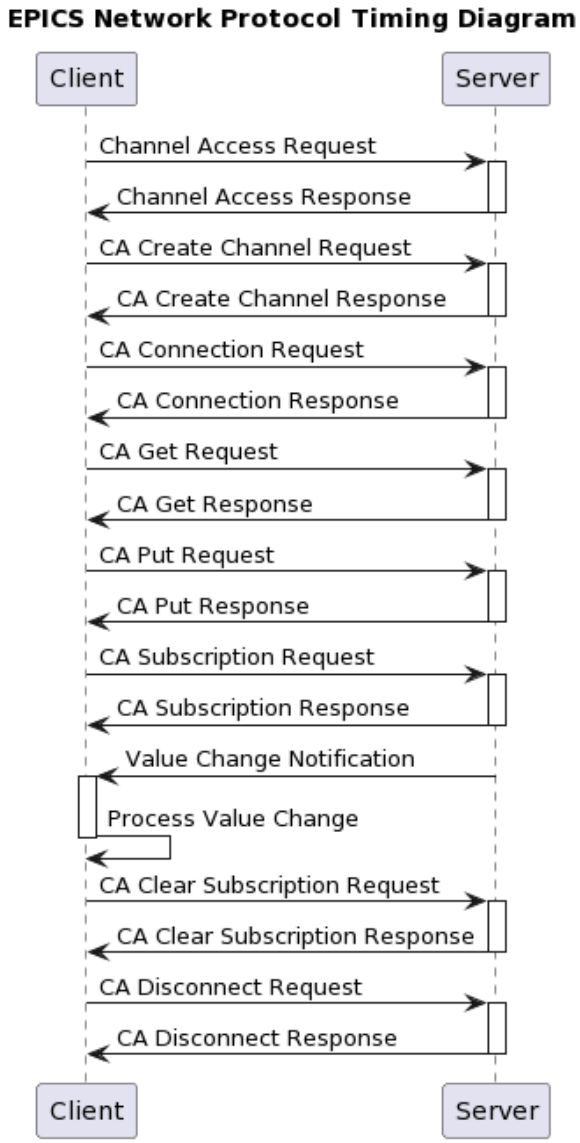


Figure 3 - Example EPICS Timing Diagram

Automatic Network Discovery

EPICS Channel Access uses network-based discovery to locate IOCs without requiring explicit IP address configuration.

When an EPICS client requests a Process Variable (PV), it sends a UDP broadcast containing the PV name. All IOCs on the broadcast domain receive the request and check whether they host the requested PV. An IOC that matches responds with its network address.

After discovery, the client establishes a direct TCP connection to the responding IOC for data transfer and monitoring. Subsequent communication occurs over TCP.

Discovery behavior characteristics:

- Uses UDP broadcast for PV name resolution
- Operates only within the local broadcast domain unless gateways are used
- Does not require preconfigured IOC IP addresses
- Automatically adapts to device IP address changes

Network Considerations

Most modern network security configurations block or restrict UDP broadcast traffic by default. As a result, EPICS automatic discovery often fails on segmented, routed, or security-hardened networks unless explicit firewall rules or routing exceptions are added to permit Channel Access broadcast traffic.

In such environments, EPICS gateways, static address configuration, or alternative communication mechanisms may be required.

Using EPICS Across Network Barriers

EPICS Channel Access is designed for use on a flat or lightly segmented network. Communication across firewalls or between isolated subnets requires additional configuration or intermediary services.

Channel Access Gateway (CAG)

The Channel Access Gateway proxies Channel Access traffic between EPICS clients and IOCs on different networks.

- Clients connect to the gateway instead of directly to the IOC
- The gateway forwards PV requests to the appropriate IOC
- Commonly uses TCP ports 5064 and 5065

Channel Access over SSH (CASSH)

CASSH uses SSH port forwarding to tunnel Channel Access traffic between networks.

- Requires an SSH server on an intermediate host
- Clients establish SSH tunnels to access remote IOCs
- No firewall port exposure beyond SSH is required

EPICS Gateway Application (GATE)

The EPICS Gateway Application provides proxy-style communication between two EPICS networks.

- Operates as an EPICS-aware intermediary
- Requires configuration of gateway endpoints and ports
- Typically uses TCP ports 2064 and 2065

Each approach enables Channel Access communication across network boundaries but requires additional components and configuration beyond a standard EPICS deployment.

EPICS CA Address Configuration

The `EPICS_CA_ADDR_LIST` environment variable is used in EPICS Channel Access (CA) to specify the IP addresses or broadcast addresses where EPICS IOCs (Input/Output Controllers) are running. This guide provides step-by-step instructions for configuring this variable on different operating systems.

Why is it needed?

- When an EPICS client (e.g., `caget`, `camonitor`) tries to communicate with an IOC, it needs to know where to look.
- If `EPICS_CA_ADDR_LIST` is not set, the system relies on automatic network scanning, which may not work in some environments (e.g., multi-subnet networks).

What values can it take?

- A single IP address (e.g., `192.168.100.20` for a specific IOC).
- A broadcast address (e.g., `192.168.100.255` for all IOCs on a subnet).
- Multiple addresses separated by spaces (e.g., `192.168.100.255 10.0.0.255`).

Linux / macOS (bash or zsh)

To temporarily set the variable, open a terminal and enter:

```
export EPICS_CA_ADDR_LIST="192.168.100.20"
```

To make it permanent edit your shell configuration file (`~/.bashrc` or `~/.zshrc`):

```
echo 'export EPICS_CA_ADDR_LIST="192.168.100.20"' >> ~/.bashrc
source ~/.bashrc
```

Windows (Command Prompt / PowerShell)

To temporarily set the variable, open a Command Prompt (cmd.exe) and enter:

```
set EPICS_CA_ADDR_LIST=192.168.100.20
```

Or PowerShell:

```
$env:EPICS_CA_ADDR_LIST = "192.168.100.20"
```

To make it permanent:

1. Open System Properties (`Win + R`, type `sysdm.cpl`, press Enter).
2. Go to Advanced → Click Environment Variables.
3. Under "System Variables", click New.
4. Enter:
   - Variable Name: `EPICS_CA_ADDR_LIST`
   - Variable Value: `192.168.100.20`
5. Click OK, then restart your terminal or system.

Additional Configuration Variables

Variable	Purpose
`EPICS_CA_AUTO_ADDR_LIST=NO`	Disables automatic interface scanning (forces manual entry).
`EPICS_CA_SERVER_PORT=5064`	Defines the Channel Access communication port.

Choosing EPICS for Your Control System

EPICS support is provided primarily for compatibility with existing EPICS-based control systems. A device whose configuration includes an `epics_server` node operates as an EPICS IOC and exposes parameters as PVs, with no custom driver.

For new control system designs, EPICS is generally not recommended. It introduces significant infrastructure, legacy design constraints, and operational complexity that are unnecessary for most modern systems. New projects should prefer IGX's more direct and lightweight communication interfaces unless EPICS compatibility is a hard requirement.

EPICS should be selected only when integration with an existing EPICS environment is required or when interoperability with EPICS-only tooling is unavoidable.

3.5.2 IGX EPICS Interface

An IGX device operates as an EPICS Input/Output Controller (IOC) when its configuration includes an `epics_server` node. It then exposes device data and control points directly to EPICS clients, with no custom EPICS driver or support module. A device whose configuration has no `epics_server` node does not run this server.

The IGX IOC hosts a set of Process Variables (PVs) that map directly to internal device parameters. PVs represent readable and writable values such as measurements, control settings, and status indicators.

By operating as a native IOC, IGX devices can be integrated into existing EPICS systems using standard client tools and workflows, with no additional EPICS-side development required.

Process Variables

PV names are automatically generated from the device's internal field paths. Each field is exposed as a corresponding EPICS PV using a deterministic naming scheme. PVs support standard EPICS data types and access semantics.

Communication

The IGX IOC communicates with EPICS clients using the Channel Access (CA) protocol. It supports normal EPICS operations including connection management, read/write requests, and subscription-based updates.

Handling Multiple IGX Devices

When multiple IGX devices of the same type are present on the network, each device must be uniquely identified to ensure correct PV addressing. IGX supports device scoping by prepending a device identifier to the channel path.

Supported identifiers include:

- IP address
- Device serial number
- Hostname

The identifier is prepended to the PV path using the format:

```
IDENTIFIER:/device/sub_module/voltage/value
```

All of the following PV names refer to the same internal channel on a single device:

```
/device/sub_module/voltage/value
IP:/device/sub_module/voltage/value
SERIAL:/device/sub_module/voltage/value
HOSTNAME:/device/sub_module/voltage/value
```

When only one IGX device is present, the unqualified path may be used. When multiple devices are present, an identifier is required.

Example: Multiple Devices Using IP Addresses

```
192.168.0.5:/device/sub_module/voltage/value
192.168.0.6:/device/sub_module/voltage/value
```

Example: Multiple Devices Using Hostnames

```
MY-DEVICE-1:/device/sub_module/voltage/value
MY-DEVICE-2:/device/sub_module/voltage/value
```

Identifier Selection

Using hostnames or serial numbers is recommended in environments where IP addresses may change due to DHCP, network reconfiguration, or device replacement. This provides stable PV naming without requiring EPICS-side changes when network assignments change.

IGX PV Alias Map File

In addition to the default PV names generated automatically by IGX, users may define additional PV aliases using a PV Alias Map XML file.

The alias map defines one or more alternate PV names and maps each to an existing IGX IO field path. An IO field may have multiple aliases. The original automatically generated PV names remain valid and are unaffected.

XML File Format

The alias map uses a simple XML structure. Each `<alias>` element specifies:

- `pv` : the alias PV name to expose
- `io` : the internal IO field path to map to

Example:

```
<?xml version="1.0" encoding="utf-8"?>
<map>
    <alias pv="mydevice:heartbeat" io="/heartbeat/value"/>
    <alias pv="{serial}_version" io="/admin/version_int/value"/>
</map>
```

In the example above:

- The alias `mydevice:heartbeat` maps to `/heartbeat/value`
- Both PVs reference the same underlying field

The following commands return identical results:

```
caget mydevice:heartbeat
caget /heartbeat/value
```

Alias PV names may include template variables. These variables are resolved at runtime using device-specific values.

Supported variables include:

- `{serial}` - Replaced with the device serial number.

Example:

```
{serial}_version
```

If the device serial number is `12345`, the resolved PV name becomes:

```
12345_version
```

Template variables allow a single alias map file to be reused across multiple devices while preserving unique PV names.

Alias XML Installation and Updating

To install or update the PV Alias Map file:

1. Navigate to the **Network** settings tab.
2. Locate the **PV Alias Map** upload field.
3. Select the XML file and upload it.

The current alias map may be downloaded using the adjacent download button.

Changes take effect immediately and do not require a device restart. Some EPICS clients may need to reconnect or restart to discover newly added PVs.

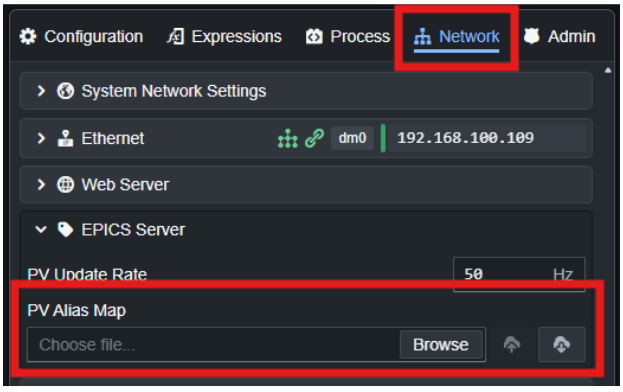


Figure 4 - The IGX PV XML file upload section

3.5.3 EPICS Utility Programs

EPICS provides several command-line utilities for interacting with PVs, including `caget`, `caput`, and `camonitor`.

The `caget` utility is used to read the current value of a PV. It accepts one or more PV names as arguments and returns their current values. For example, to read the value of a PV named `my_pv`, you can use the command `caget my_pv`.

The `caput` utility is used to write a new value to a PV. It accepts two arguments: the PV name and the new value. For example, to set the value of a PV named `my_pv` to `5.0`, you can use the command `caput my_pv 5.0`.

The `camonitor` utility is used to continuously monitor a PV for changes and print them to the console. It accepts one or more PV names as arguments and continuously updates their values on the console. For example, to monitor a PV named `my_pv`, you can use the command `camonitor my_pv`.

These utilities are typically installed as part of the EPICS Base distribution. To install them, you need to download and install the EPICS Base distribution for your platform. Once installed, the utilities should be available on the command line.

Getting the Current Hostname

To get the current hostname of an IGX device, you can use the `caget` utility to read the corresponding Process Variable (PV) value. The PV for the hostname is located at /net/hostname/value. To read the current value, enter the following command in a terminal window:

```
caget /net/hostname/value
```

This command will output the current hostname of the device. For example:

```
192.168.1.100:/net/hostname/value     MY-DEVICE
```

In this example, the current hostname of the device is "MY-DEVICE".

Setting a New Hostname

To set a new hostname for an IGX device, you can use the caput utility to write the new value to the corresponding PV. The PV for the hostname is located at `/net/hostname/value`. To set a new value, enter the following command in a terminal window:

```
caput /net/hostname/value NEW-HOSTNAME
```

Replace "NEW-HOSTNAME" with the desired new hostname for the device. After entering this command, the new hostname value will be written to the device, and it will become accessible via the new hostname.

Note that changing the hostname of an IGX device may require updating the IP address associated with the device in your network configuration, depending on your network setup.

Monitoring Heartbeat

In the example below, we use the `camonitor` utility to continuously monitor the `/heartbeat/value` Process Variable on an IGX device. The `/heartbeat/value` variable is a Boolean type, and it alternates between `true` and `false` at a frequency of 1 Hz.

The `camonitor` command continuously queries the value of the PV and prints the updated value to the terminal as soon as it changes. This allows you to monitor the state of the variable in real-time and react to changes as necessary.

To monitor the `/heartbeat/value` Process Variable, use the following command:

```
camonitor /heartbeat/value
```

This will print the updated value of the `/heartbeat/value` variable to the terminal as it changes, similar to the following:

```
/heartbeat/value                false
/heartbeat/value                true
/heartbeat/value                false
/heartbeat/value                true
/heartbeat/value                false
/heartbeat/value                true
/heartbeat/value                false
/heartbeat/value                true
/heartbeat/value                false
/heartbeat/value                true
/heartbeat/value                false
```

3.5.4 EPICS Python Examples

This programming guide demonstrates how to access field values on an IGX device using EPICS in Python. Before you begin, ensure you have the following prerequisites:

1. Python 3.x installed.
2. EPICS Base and the Python epics library (`pyepics`) installed.
   - To install `pyepics` , run `pip install pyepics` .

Python Library Overview

`pyepics` is a Python library that provides an easy-to-use interface to the Experimental Physics and Industrial Control System (EPICS), a set of software tools for building distributed control systems. `pyepics` allows Python developers to interact with process variables (PVs) and monitor them for changes.

I/O Functions

The `pv_get()` function is used to read the current value of a PV, and the `pv_put()` function is used to write a new value to a PV. The functions are just convenient wrappers for the channel functions; they are helpful for writing simple scripts.

- `pv_get(pvname, as_string=False, as_numpy=False)` : Get the current value of the specified PV. Returns a string or a `numpy` array, depending on the value of `as_string` and `as_numpy` .
- `pv_put(pvname, value, wait=True)` : Write the specified value to the PV. If `wait` is True (default), wait for the write operation to complete before returning.

Channel Functions

Channels are used to represent PVs in the `pyepics` library. The following functions are used to create, connect to, and manipulate channels:

- `create_channel(pvname, callback=None, connect=True)` : Create a channel object representing the specified PV. If `connect` is True (default), connect to the PV immediately. If `callback` is provided, call the specified function when the connection is established.
- `connect_channel(chid, timeout=None)` : Connect to the specified channel. If `timeout` is provided, wait for the specified number of seconds for the connection to be established.
- `get(chid, as_string=False, as_numpy=False)` : Get the current value of the specified channel. Returns a string or a `numpy` array, depending on the value of `as_string` and `as_numpy` .
- `put(chid, value, wait=True)` : Write the specified value to the channel. If `wait` is True (default), wait for the write operation to complete before returning.

- `disconnect_channel(chid)` : Disconnect from the specified channel.
- `destroy_channel(chid)` : Destroy the specified channel object.

Subscription Functions

Subscriptions are used to listen for changes to PVs. The following functions are used to create and manage subscriptions:

- `create_subscription(chid, callback=None)` : Create a subscription object for the specified channel. If `callback` is provided, call the specified function when the value of the channel changes.
- `clear_subscription(eventID)` : Remove the specified subscription.

Other Functions

The following functions are used for miscellaneous tasks:

- `poll(evt=None, iot=None)` : Wait for EPICS events and I/O to occur. If `evt` is specified, wait for the specified number of seconds for an event to occur. If `iot` is specified, wait for the specified number of seconds for I/O to occur.
- `ca.initialize_libca()` : Initialize the Channel Access library.
- `ca.finalize_libca()` : Finalize the Channel Access library.
- `ca.flush_io()` : Flush pending I/O operations.

Getting a PV (IO) Value

Start by importing the `epics` library, which provides the necessary functions and classes for interacting with EPICS PVs.

```
import epics
```

Create an EPICS Process Variable (PV) object using the field path of the value you wish to access on the IGX device. In this example, we will access the heartbeat value located at `/heartbeat/value`.

```
pv = epics.PV("heartbeat/value")
```

Use the `get()` method of the PV object to retrieve the current value of the specified field.

```
value = pv.get()
```

The complete Python script for accessing field values on an IGX device using EPICS is shown below:

```
import epics

# Create a PV object for the desired field path
pv = epics.PV("/heartbeat/value")

# Retrieve the current value of the field
value = pv.get()

# Print the retrieved value to the console
print(value)
```

This script demonstrates a straightforward way to read field values from an IGX device using EPICS in Python. You can modify the field path in the `epics.PV()` constructor to access other values as needed.

Putting a PV (IO) Value

This Python script sets the hostname of an IGX device using the `put` function in EPICS.

```
import epics
```

```
# Create a PV object for the desired field path
hostname_pv = epics.PV("/net/hostname/value")

# Set a new hostname for the IGX device
new_hostname = "MY-DEVICE"
hostname_pv.put(new_hostname)

# Retrieve the updated hostname value
updated_hostname = hostname_pv.get()

# Print the updated hostname to the console
print("Updated hostname:", updated_hostname)
```

In this script, we start by importing the `epics` library. Next, we create a PV object for the desired field path (`/net/hostname/value`) by calling the `epics.PV()` constructor. This field path corresponds to the hostname of the IGX device.

We then set a new value for the hostname using the `put` function, passing in the desired new hostname as a string (in this case, `"MY-DEVICE"`). After setting the hostname, we retrieve the updated hostname value from the device using the `get` function and print the updated hostname to the console.

This script demonstrates a simple way to set field values on an IGX device, specifically the hostname, using the `put` function with EPICS in Python. You can modify the field path in the `epics.PV()` constructor and the `new_hostname` variable to set other field values as needed.

Zeroing a T1 Probe

This example demonstrates how to zero a probe by setting the device offset to the current field measurement using EPICS in Python. This is a common procedure before performing a relative measurement.

```
import epics
import time

# Create our PV objects
field = epics.PV("/t1/probe/average_field/value")
offset = epics.PV("/t1/probe/offset/value")

print("Zeroing field probe")

# First, we get rid of any existing offset by setting it to zero and waiting
offset.put(0.0)

# Wait for the new offset to propagate to the new data
time.sleep(0.5)

# Get the current field.
# Set the offset to the previously measured field, effectively zeroing it.
offset.put(field.get())

# Wait for the new offset to propagate to the new data
time.sleep(0.5)

# Get the field again to confirm the zeroing worked.
print("Newly zeroed field", field.get(), "G")
```

The script consists of the following steps:

1. Import the required libraries, `epics` for communicating with the EPICS server and `time` for introducing delays.
2. Create two PV objects: `field` for the average field value and `offset` for the probe's offset value. Both PVs are constructed using their respective field paths.
3. Print the message "Zeroing field probe" to indicate the start of the zeroing process.
4. Set the current offset value to 0.0 using the `put` function, effectively clearing any existing offset.
5. Wait for 0.5 seconds using `time.sleep` to allow the new offset value to propagate through the system.
6. Get the current field value using the `get` function and set the offset value to the retrieved field value. This step effectively zeros the probe by compensating for the current field value.
7. Wait for another 0.5 seconds using `time.sleep` to let the new offset value propagate through the system.
8. Get the field value again using the `get` function to confirm that the zeroing process was successful.
9. Print the newly zeroed field value.

In production code, it is advisable to create wrapper functions to encapsulate the EPICS communication logic, which helps reduce code complexity and enhances readability.

Polling Heartbeat

One way to get continuous data is to periodically poll EPICS PVs. The following example shows how to do this for the `/heartbeat/value` IO.

```
import epics

def on_value_change(pvname=None, value=None, **kw):
    print(pvname, value)

pv = epics.PV('/heartbeat/value')
subscription = pv.add_callback(on_value_change)

while True:
    epics.ca.poll()
```

The `on_value_change` function is defined to print the name and value of the PV whenever its value changes.

The PV object is created with the `epics.PV()` constructor and its value changes are monitored with `pv.add_callback(on_value_change)`.

The `while` loop calls the `epics.ca.poll()` function to check for new PV values and invoke the `on_value_change` function when a change is detected. The `poll()` function checks for new events in the channel access event queue and calls any associated callbacks. The loop runs indefinitely, allowing the program to continuously monitor the PV.

In this example, the `/heartbeat/value` field is changing between `1` and `0` at a rate of 1 Hz, so the output alternates between these two values. The output of the program will look something like the following:

```
/heartbeat/value 1
/heartbeat/value 0
/heartbeat/value 1
/heartbeat/value 0
/heartbeat/value 1
/heartbeat/value 0
/heartbeat/value 1
...
```

Subscribing to Heartbeat

In the `pyepics` subscription system, a channel is monitored for changes using a subscription. A subscription is an object that listens for changes to a channel and calls a user-defined function (the callback function) when a change occurs.

When you create a subscription object, you provide it with a channel object and a callback function. The subscription object then monitors the channel for changes and calls the callback function whenever a change occurs.

Subscriptions can be created and destroyed dynamically, which makes them very flexible. You can create as many subscriptions as you need, and you can customize the behavior of each subscription by providing a different callback function.

This example uses the EPICS channel access (CA) library to connect to the PV `/heartbeat/value` and monitor changes to its value.

```
import epics  # Import the epics module
import signal  # Import the signal module
import sys  # Import the sys module


# Define a function to be called when a connection is made
def onConnect(pvname=None, **kw):
    print('on Connect %s %s' % (pvname, repr(kw)))


# Define a function to be called when a value changes
def onChanges(chid=None, value=None, **kw):
    print('on Change chid=%i value=%s' % (int(chid), repr(value)))


# Define a signal handler to exit the program gracefully
def signal_handler(signal, frame):
    print("Exiting program...")
    sys.exit(0)


# Create a channel to monitor the heartbeat value
chid = epics.ca.create_channel('/heartbeat/value', callback=onConnect)

# Connect to the channel
epics.ca.connect_channel(chid)

# Create a subscription to receive notifications of changes to the channel
eventID = epics.ca.create_subscription(chid, callback=onChanges)

# Set up the signal handler to exit the program gracefully on Ctrl+C
signal.signal(signal.SIGINT, signal_handler)

# Poll for events and I/O every 0.025 seconds and 5 seconds, respectively
while True:
    epics.poll(evt=0.025, iot=5.0)
```

The `onConnect()` function is called when a connection is established to the channel. It prints a message indicating that a connection has been made and the details of the connection.

The `onChanges()` function is called whenever the value of the channel changes. It prints a message indicating the new value and the details of the change.

The `signal_handler()` function is used to exit the program gracefully when the user presses `Ctrl+C` on the keyboard.

The `create_channel()` function creates a new channel object that represents the PV. The `connect_channel()` function connects to the PV and starts monitoring it for changes.

The `create_subscription()` function creates a subscription object that listens for changes to the PV. When a change occurs, the `onChanges()` function is called.

The `poll()` function is used to wait for events and I/O to occur. It waits for a short period of time (0.025 seconds) and then checks for new events or I/O. If there are any, it handles them. This loop runs indefinitely until the program is terminated.'

The program output should look something like this:

```
on Connect /heartbeat/value {'chid': 2015005851712, 'conn': True}
on Change chid=2015005851712 value=1
on Change chid=2015005851712 value=0
on Change chid=2015005851712 value=1
on Change chid=2015005851712 value=0
on Change chid=2015005851712 value=1
on Change chid=2015005851712 value=0
Exiting program...
```

3.6 IGX SFTP Protocol Guide

3.6.1 Use SFTP for IGX IO Data

Secure File Transfer Protocol (SFTP) is a widely used protocol for securely transferring files over a network. Integrating SFTP with IGX systems allows users to efficiently transfer IO data while ensuring data confidentiality and integrity during transmission. This document outlines the process of using SFTP for IO file transfer in IGX environments.

Accessing IGX IO Data via SFTP

By configuring an SFTP server on the IGX system, users can securely access and transfer IO data. To facilitate this, IGX should expose its IO data as files within a designated directory (e.g., `/io`). Clients can then connect to the SFTP server using an SFTP client, browse the `/io` directory, and transfer files as needed.

To ensure consistency and ease of access, the IO files within the `/io` directory should adhere to a standardized naming convention and path structure. This convention should be uniform across various communication protocols, making it easier for users to locate and access specific IO data.

For example, if the heartbeat value field JSON file is located at `/io/heartbeat/value.json` in other communication protocols, the same path should be used when accessing the file via SFTP.

Python Example

To connect to an IGX SFTP server programmatically, you can use an SFTP library in your preferred programming language. In this example, we will use Python and the Paramiko library, which is a widely used library for SSH and SFTP connections.

First, make sure you have the Paramiko library installed. You can install it using pip:

```
pip install paramiko
```

Now, you can use the following code snippet to connect to the IGX SFTP server using the default port number (22), username (root), and password (root):

```
import paramiko
```

```
# Define SFTP server connection details
hostname = 'your_igx_server_ip_or_hostname'
port = 22  # Default SFTP port
username = 'root'
password = 'root'

# Create an instance of the SSH client
ssh_client = paramiko.SSHClient()

# Automatically add the server's public key (this should be done with caution in
production)
ssh_client.set_missing_host_key_policy(paramiko.AutoAddPolicy())

# Connect to the SFTP server
ssh_client.connect(hostname, port, username, password)

# Open an SFTP session
sftp = ssh_client.open_sftp()

# Interact with the server's file system (e.g., list files in the /io directory)
files = sftp.listdir('/io')
print("Files in /io directory:", files)

# Close the SFTP session and SSH connection
sftp.close()
ssh_client.close()
```

Replace `'your_igx_server_ip_or_hostname'` with the actual IP address or hostname of your IGX server. This code snippet demonstrates connecting to the IGX SFTP server, listing the files in the `/io` directory, and closing the connection. You can extend the script to perform additional tasks, such as uploading or downloading files, based on your requirements.

Please note that using the username 'root' and password 'root' in a production environment is not recommended due to security concerns. In a real-world scenario, you should use a secure authentication method, such as key-based authentication, and follow the principle of least privilege by using a dedicated user account with restricted permissions.

3.7 IGX Qnet Protocol Guide

3.7.1 Qnet Overview

QNX Qnet is a distributed, transparent networking system designed for real-time applications. Developed by QNX Software Systems, it is an integral part of the QNX Neutrino Real-Time Operating System (RTOS). Qnet enables devices to discover each other, communicate, and share resources in a highly efficient and predictable manner. This overview will provide insights into the technical aspects of Qnet, including device discovery, network packet passing, and file exposure between nodes.

Network Protocol

The QNX Qnet system does not rely on standard network protocols or specific port numbers for its device discovery process. Instead, it uses a custom lightweight protocol built on top of the Transparent Interprocess Communication (TIPC) protocol, which is designed specifically for QNX Neutrino RTOS environments.

The device discovery process in Qnet is based on link-local multicast mechanisms. QNX Neutrino nodes send out node advertisements containing their hostname, IP address, and available services when they join the network. These advertisements are multicast to a predefined multicast address reserved for Qnet.

While Qnet does not use traditional port numbers as in the case of TCP/IP-based protocols, it does employ "service ranges" to identify and communicate with different services on a node. Service ranges in TIPC are similar to port numbers in other protocols, providing unique identifiers for various services running on a node.

Device Discovery

Qnet utilizes a serverless approach to automatically discover devices on the network. This is achieved by employing a link-local multicast mechanism that allows devices to identify themselves and their available services without the need for a centralized server. When a QNX Neutrino node is connected to a network, it broadcasts a "node advertisement" to the network, which includes information about its hostname, IP address, and available services. Other devices on the network receive this advertisement and update their internal routing tables accordingly. The node discovery process is continuous, allowing devices to join and leave the network dynamically.

The Net Directory

The Qnet `/net` directory is an essential component of the QNX Neutrino RTOS Qnet protocol that allows transparent distributed processing across multiple QNX nodes. The `/net` directory serves as a virtual mount point for remote file systems and processes, enabling seamless access to resources on other QNX systems in the network.

When a QNX system with Qnet enabled wants to access resources on a remote QNX node, it can use the `/net` directory to browse and interact with the remote file system as if it were local. The `/net` directory provides a unified namespace for all connected nodes, making it easy to navigate and access resources across the entire network.

Here's a brief explanation of how the `/net` directory works:

1. Discovery: When Qnet-enabled QNX nodes start up, they broadcast their presence over the network. Neighboring Qnet nodes receive these broadcasts and automatically establish connections with each other. This process creates a network of interconnected QNX nodes, all of which are accessible through the `/net` directory.
2. Accessing Remote Resources: To access a resource on a remote QNX node, the local QNX system needs to reference the resource using the `/net` directory. The path format is `/net/remote_node_name/resource_path`. For example, to access the `/tmp` directory on a remote node named `node2`, the local system would use the path `/net/node2/tmp`.
3. Remote Process Execution: In addition to accessing remote file systems, the `/net` directory also allows for transparent remote process execution. When a process is launched with a pathname that begins with `/net/remote_node_name`, the process runs on the specified remote node but appears to be local from the perspective of the initiating system. This feature is useful for load balancing and distributed processing across multiple QNX nodes.
4. Resource Sharing: Qnet's `/net` directory makes it possible to share resources, such as file systems, devices, and processes, among QNX nodes in a transparent and efficient manner. This resource sharing capability simplifies the development of distributed applications and enables more effective system management.

Network Packet Passing

Qnet's efficient network packet passing is achieved through a lightweight, connection-oriented protocol called Transparent Interprocess Communication (TIPC). This protocol is designed to provide low-latency, reliable communication between processes on the same or different nodes.

TIPC operates at the transport layer, using a combination of connection-oriented (stream) and connectionless (datagram) services to transmit data. This allows Qnet to adapt to varying application requirements, such as handling high-frequency real-time data or streaming large files.

In order to minimize overhead, Qnet leverages zero-copy techniques, which eliminate the need for data copying between user space and kernel space. Instead, data is transferred directly from the

sender's buffer to the receiver's buffer, reducing the number of memory operations and, consequently, the communication latency.

File Exposure between Nodes

Qnet's transparent networking enables seamless access to files and resources on different nodes as if they were located on the local node. The QNX Neutrino RTOS provides a unified file system namespace, which allows applications to access remote files using standard file system operations.

When an application attempts to access a file on a remote node, the local node's file system layer sends a request to the remote node's Qnet server via TIPC. The remote node processes the request and sends the required data back to the local node. This transparency allows developers to create distributed applications without the need for specialized APIs or programming paradigms.

3.7.2 Integrating Qnet with IGX

IGX leverages Qnet's transparent networking capabilities to efficiently manage and expose its IO data through a virtual file system. By creating special files under the `/io` directory, IGX allows clients connected to the Qnet system to easily interact with IO data by reading and writing these files. The dynamic nature of these files ensures that the contents are updated based on the IO values in real-time. Moreover, these files are not stored on non-volatile memory systems like traditional files, but instead exist solely for facilitating IO data access and manipulation.

Accessing IO Data through Qnet

Clients can access IGX IO data via the Qnet system by reading and writing to the special files located under the `/io` directory. Since Qnet provides a unified file system namespace, these files can be accessed as if they were present on the local system. This transparent access to IO data simplifies the interaction with IGX and eliminates the need for any specialized APIs.

File Path Conventions in IGX

IO files within the `/io` directory adhere to standardized path conventions, which are identical to those used in other IGX network protocols, such as HTTP. This consistency in path naming simplifies the process of locating and accessing specific IO data across various communication protocols.

For instance, the heartbeat value is the file `/io/heartbeat/value.json`. Its body is the JSON value itself, such as `false`, not an object wrapping that value. Writing a JSON value to the same file sets the field, for a field that is writable.

3.7.3 Python Qnet Example

This guide demonstrates how to create a Python script that runs on a QNX Qnet node and reads the `/io/heartbeat/value.json` file on another Qnet node. To follow this guide, you should have Python installed on your QNX system and have a basic understanding of the Qnet protocol.

Requirements

- A Qnet-enabled QNX network with at least two connected nodes

Preparing the Environment

Ensure that the Qnet nodes are properly configured and that your local node can access the remote node's file system through the `/net` directory. Verify this by browsing the remote node's file system from the local node:

```
ls /net/remote_node_name
```

Replace `remote_node_name` with the hostname or IP address of the remote QNX node.

Creating the Python script

Create a new Python script named `read_remote_io.py` using your preferred text editor. Add the following code to the script:

```
import json
import sys

def read_remote_heartbeat(remote_node_name):
    remote_file_path = f"/net/{remote_node_name}/io/heartbeat/value.json"

    try:
        with open(remote_file_path, "r") as file:
            # The file body is the JSON value itself, for example false,
            # not an object with a "value" key.
            value = json.load(file)
            print(f"Heartbeat value on {remote_node_name}: {value}")
    except FileNotFoundError:
        print(f"Error: File not found on remote node '{remote_node_name}'.")
    except Exception as e:
        print(f"Error: {e}")

if __name__ == "__main__":
    if len(sys.argv) < 2:
        print("Usage: python read_remote_io.py <remote_node_name>")
    else:
        remote_node_name = sys.argv[1]
        read_remote_heartbeat(remote_node_name)
```

This script defines `read_remote_heartbeat`, which reads `/net/<node>/io/heartbeat/value.json` on the remote node. The file contains the field's JSON value directly, the same bytes HTTP returns for that path. `json.load` therefore returns `false` or `true` for the heartbeat, not a dictionary.

Running the Python script

Save the `read_remote_io.py` script and run it on the local QNX node:

```
python read_remote_io.py <remote_node_name>
```

Replace `<remote_node_name>` with the hostname or IP address of the remote QNX node. The script will read the `/io/heartbeat/value.json` file on the remote node and print the heartbeat value.

In case of any errors, such as the file not being found or an issue with the JSON data, the script will print an appropriate error message.

4 IGX Standard IO Interfaces

4.1 IGX Button IO Interface

4.1.1 Overview

In IGX every interface is expressed as IO, including commands. This lets programmers use the same interface to set configurable parameters and to initiate commands, instead of separate interfaces for configuration and remote procedure calls. IGX calls these button IO, after the graphical widget shown in the web browser.

Button IO is a form of digital IO that represents an action rather than a persisted setting. Writing a low (false)→high (true) transition triggers the action. What happens next depends on the button variant: some reset to false immediately; others remain high until the client sets them back to false. See Button Variants below for details.

4.1.2 Button Variants

All button IO share the same low-to-high trigger mechanism, but there are three variants with different reset behavior:

Type	Behavior
`ButtonIO`	The default and most common variant. The device automatically resets the IO back to false immediately after a client sets it to true, regardless of what the client does. From a client's perspective this makes every trigger a single momentary pulse; there's no need to write false afterward.
`ConfirmButtonIO`	Identical to `ButtonIO` on the wire; a client triggers it exactly the same way. The only difference is the web GUI shows a confirmation prompt before the action runs. Programmatic clients that set it directly bypass that prompt, the same way a script bypasses any other GUI-only convenience.
`HoldButtonIO`	Does not auto-reset. Setting it to true starts a continuous effect that lasts until the client explicitly sets it back to false; both the true and false transitions emit their own event, since press and release are both meaningful. Used for actions that should only continue while a control is actively held.

4.1.3 Triggering a Button

Over WebSocket, a button is addressed like any other field: its vertex path with no `/io` prefix and a trailing `/value`, for example `path = "/fx4/data_acquisition/collect_button/value"`. An example of triggering a standard `ButtonIO` or `ConfirmButtonIO`:

```
# Set the button to the high state in order to trigger the event.
send("set", {path: True})

# No need to set it back to false yourself: the device does this
# automatically right after the transition is processed.
```

It's still good practice to write false before true if you're unsure of a button's current state (for example, if a previous, nonconforming client left it high), since a button already at true will not re-trigger on another true write.

```
# Defensive reset, only needed if you suspect the button may already be high.
send("set", {path: False})
send("set", {path: True})
```

A `HoldButtonIO` needs an explicit release, since it will not reset itself:

```
# Start the continuous effect.
send("set", {hold_path: True})

# ... later, when the effect should stop ...
send("set", {hold_path: False})
```

If two clients attempt to transition the same button into the high state at the same time, only one event will trigger. This is an intentional behavior of IGX and protects the system from race conditions. If you need arbitrary triggering between many clients, subscribe to the button's state over the WebSocket interface and write your code so your client waits for other clients to finish with the button before triggering it itself.

4.1.4 Python HTTP Example

Buttons can also be triggered with a plain HTTP PUT, the same way as any other writable IO (see the IGX HTTP Protocol Guide (see page 13) for the general pattern):

```
import requests, json

url = "http://" + ip + "/io/fx4/data_acquisition/collect_button/value.json"

# Trigger the button with a true.
print(requests.put(url, json.dumps(True)).json())

# A standard ButtonIO has already reset itself to false by the time this
# request completes, so there's normally no need to PUT false afterward.
```

4.1.5 Hidden "Spinning" Indicator

Every button also has a `spinning` child IO, hidden from the standard web GUI tree but still addressable the same way as any other field, for example `.../button_name/spinning/value` over WebSocket or `.../button_name/spinning/value.json` over HTTP. A component can set it to true while the action the button represents is still in progress, which causes the web GUI to show a spinning icon on that button instead of its usual icon. Custom clients can subscribe to it the same way to reflect an in-progress state in their own UI without having to infer it from other IO.

4.1.6 Real-World Example: Data Acquisition Table Controls

The Data Acquisition table's `collect_button`, `cancel_button`, and `clear_button` (documented in the IGX Data Acquisition Table API Guide (see page 80)) are all plain `ButtonIO`, so triggering any of them is exactly the pattern above: write true, and the device resets it to false on its own.

```
# path is the table's own vertex path, e.g. "/fx4/data_acquisition"
# Clear old data, then arm a new capture. No need to reset either
# button back to false afterward; both are plain ButtonIO.
send("set", {path + "/clear_button/value": True})
send("set", {path + "/collect_button/value": True})
```

That table also has a `universal_button`, an unlabeled `ButtonIO` meant to drive a single "Start/Stop Collection" control. Because it has no label, it doesn't appear in the generic auto-generated web GUI, but it's fully addressable and triggers the same way as any other button.

4.2 IGX High Voltage IO Interface

4.2.1 Overview

Many IGX systems utilize an integrated high voltage power supply module. In order to simplify integration and promote reuse of code, Pyramid has created a standardized IO interface that is compatible with multiple products.

High voltage modules are essential components in a variety of applications, including medical control systems, where reliable operation and fault detection are critical. These modules may feature both internal and external sense circuits, which provide an additional layer of redundancy to ensure proper functioning and connection.

This guide will cover how to manipulate and monitor the IO interface in order to work with this controller effectively through a programable interface. See IGX Network Protocols (see page 10) for more information about the fundamentals of how to interface with IO themselves.

Interface IO

To use the high voltage module interface pragmatically, you can use the following IO. They can be used in different and flexible combinations to suite your particular application.

Typically, the high voltage component will have a parent path like: `/<device>/high_voltage/`.

Name	Detail	Type
`../state`	Corresponds to the current state of the module.	Read-only String
`../permit/user_command`	Turns on or off the power supply if the permit is granted by the interlock states.	Digital
`../command_voltage`	Sets the command voltage for the power supply output. Units are in volts.	Analog
`../monitor_voltage_internal`	The output voltage as measured by the internal sense circuit directly on the output. Units are in volts.	Read-only Analog
`../monitor_voltage_external`	The detected voltage as measured by the external sense circuit. Units are in volts.	Read-only Analog

Module States

The `state` IO represents the state of the module using a short and pain-English string.

State	Description
ready	HV module is off, but ready to be enabled and all interlocks are satisfied.
enabled	HV module is enabled, and all interlocks are satisfied.
warning	HV module is enabled, and there is at least one interlock in warning.
fault	HV module is off, at least one interlock is in fault and preventing re-enable.

Safety Interlocks

Software interlocking can be enabled on the module preventing the module from being turned on under some circumstances. This is in order to prevent the module from damaging itself or other external equipment. All interlocks can be disabled, programmatically or through the GUI interface.

Internal Voltage Sense Circuit

The internal sense circuit is an integral part of the high voltage module, monitoring the output voltage generated by the power supply. This circuit ensures that the module is functioning correctly and providing the expected output voltage.

The feedback of this circuit may be quite slow depending on the voltage divider and filters used. Please see the corresponding product documentation for more information.

External Voltage Sense Circuit

In some high voltage modules, an external sense circuit is also available. This additional circuit is designed to monitor the voltage at the external device connected to the module. By feeding the voltage information back to the control system, it provides a means of verifying the proper functioning and connection of the external device, as well as the integrity of cables and connections.

External sense circuits play a crucial role in medical control systems where the accurate delivery of high voltage is essential. In such applications, a lack of high voltage or a disconnected cable could lead to serious harm or even fatal consequences. For example, an ion chamber may use the high voltage as a bias for an electrode, and an external sense circuit will ensure that the connection is secure and operating as expected.

It is important to note that not all high voltage modules are equipped with external sense circuits. To determine whether a specific module includes this feature, consult the product documentation or contact the manufacturer. In cases where a high voltage module does not have an external sense circuit, the `monitor_voltage_external` IO will still be available. However, this IO value will always read 0 volts, as there is no external voltage information being fed back to the control system.

4.2.2 Use Case Examples

This section provides use case examples and pseudo code to demonstrate how to interact with the IO interface for setting and reading voltage values.

Setting and Enabling High Voltage Output

In this example, the pseudo code illustrates how to set the output voltage to 100V and enable the power supply.

```
set("command_voltage", 100) # Set the output voltage to 100V
set("permit/user_command", True) # Enable the power supply
```

Reading Internal Voltage Values

After enabling the power supply, the pseudo code demonstrates how to read the internal voltage value from the high voltage module.

```
readback = get("monitor_voltage_internal") # Read the internal voltage value
print(f"The internal voltage value is: {readback} V") # Display the internal voltage
value
```

Monitoring External Voltage Values (Optional)

If your high voltage module has an external sense circuit, you can monitor the external voltage value as well. Note: If your high voltage module does not have an external sense circuit, the `monitor_voltage_external` IO will always read 0 volts. Here's the pseudo code to read the external voltage value:

```
external_readback = get("monitor_voltage_external") # Read the external voltage value
print(f"The external voltage value is: {external_readback} V") # Display the external
voltage value
```

4.3 IGX Relay IO Interface

IGX is designed to offer a standardized and uniform interface for the control and monitoring of relays. Relays are vital in safety control systems, and a uniform interface greatly aids developers by guaranteeing secure and uncomplicated implementations.

The operation of a relay, whether opening or closing, is always managed through a permit, which can be associated with a variable number of interlocks. For a detailed explanation of interlocks and permits, please refer to the IGX - User Manual. If the permit is revoked (unsafe state) then the relay will always open.

4.3.1 Basic Relay Control IO

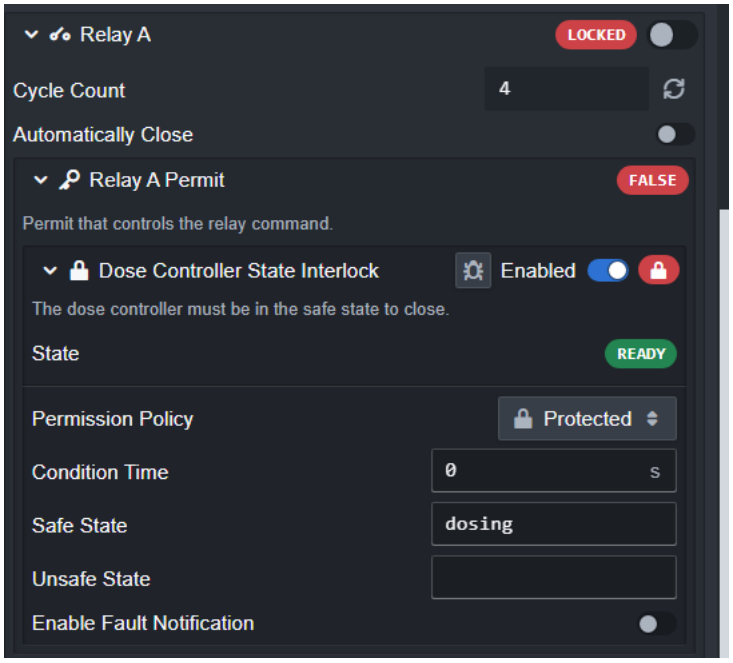
IO Path	Description
`.../permit/user_command`	Commands the relay open or closed. A true command will try to close the relay if the interlocks are granted, and false command will always open the relay.
`.../state`	The current state of the relay. Locked relays are open but cannot be closed due to an interlock. States: “opened”, “closed”, or “locked”

4.3.2 Advanced Control IO

IO Path	Description
`.../automatically_close`	When set to true, the relay will automatically close when the interlocks are granted. False by default.
`.../cycle_count`	The number of relay cycles since the last reset. Useful for tracking relay lifetime.

4.3.3 Web GUI

The Web GUI exposes the relay functionality.



The following setups and readbacks are available:

- Relay State - the relay state, “CLOSED” if the relay is closed, “OPENED” if the relay is opened, or “LOCKED” if the relay is under control by a process, such as Dose Controller.
- Relay Close - switch to enable or disable the relay if not LOCKED.
- Cycle Count - readback that counts the number of times the relay is closed.
- Automatically Close - if not LOCKED, this switch has the same effect as the Relay Close switch. In LOCKED mode
- Relay A Permit - shows the permit state, TRUE or FALSE. The relay can only be closed when this state is TRUE. If the Dosimetry Controller option is enabled, the permit state will be determined by various factors in that module.
  - Dose Controller State Interlock - enables or disables control of the relay by the Dose Controller. The permit will always be TRUE if this interlock is disabled. This interlock can only be disabled if the Dose Controller state is “READY” and the operation mode is not medical.
  - Condition Time - the amount of time in seconds that the fault condition must be true for the interlock to fault.
  - Safe State - if this state is achieved by the Dose Controller than the interlock will be enabled and the relay can be closed. If a Safe State is specified then the Unsafe State is ignored.
  - Unsafe State - if this state is achieved by the Dose Controller and the Safe State is blank, then the interlock will be disabled.
  - Enable Fault Notification - when enabled faults will be written to the Notification log.

4.4 IGX Offset Correction (Zero) IO Interface

4.4.1 Overview

Many IGX products include a software feature that enables users to correct offsets on current and voltage inputs. This feature measures the background noise over a configurable period, calculates the average, and applies it as an offset. Users can also clear or manually set this offset as needed. This functionality is particularly useful for simulating leakage conditions or implementing more advanced correction methods.

The FX4 offset correction GUI is shown here as an example. Most IGX devices have a similar interface, though the specific channels and units may vary depending on the device.

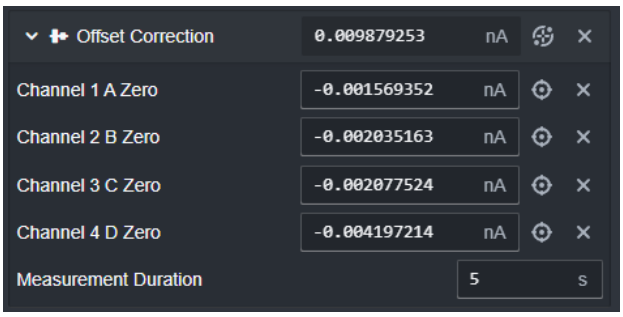


Figure 5 - FX4 Offset Correction

Interface IO

To use the offset correction module interface pragmatically, you can use the following IO. They can be used in different and flexible combinations to suite your particular application.

Typically, the offset correction component will have a parent path like: `/<device>/adc/offset_correction/`.

Name	Detail	Type
`../value`	The current offset value. Changing this value will change the offset.	Analog

Name	Detail	Type
`../sequence/start_button`	Starts the automatic sequence. Once complete the offset will be set based on the current inputs.	Button Digital
`../sequence/stop_button`	Stop a running sequence if it is started. Does nothing if the sequence has not started.	Button Digital
`../clear_button`	Sets the current offset value to zero.	Button Digital
`../collecting`	True when the module is collecting samples in order to calculate the offset, false when the offset is calculated and set.	Read-only Digital

4.5 IGX Dose Controller IO Interface

4.5.1 Overview

The Dose Controller component in IGX is capable of delivering a pre-defined aliquot of dose to a given target. It does this by measuring some given sense input that proportionally represents the dose rate or dose accumulation. It also carefully measures time elapsed, and remaining, in order to accurately stop a delivery through a flexible output interface.

The controller software is optionally capable of converting from dose units (MU, Gy, Gp, etc.) to machine units (nC). This conversion is handled by a flexible and powerful mathematical expression system called “expression models”.

This guide will cover how to manipulate and monitor the IO interface in order to work with this controller effectively through a programable interface. See IGX Network Protocols (see page 10) for more information about the fundamentals of how to interface with IO themselves.

Primary Control IO

To use the dose controller interface pragmatically, you can use the following IO. They can be used in different and flexible combinations to suite your particular application.

Typically, the dose controller will have a parent path like: `/<device>/dose_controller`.

IO Path	Description
`.../state`	READONLY STRING Corresponds to the current state of the controller.
`.../start_button`	BUTTON Start the controller if allowed by interlocks. Does nothing if interlocks are not granted.
`.../pause_button`	BUTTON Pause the controller unconditionally. Does nothing if the controller is not dosing.
`.../stop_button`	BUTTON Stops the controller unconditionally and permanently, preventing resuming the current session without an explicit reset command.
`.../reset_button`	BUTTON Reset current dose delivery session if allowed. Does nothing unless the controller is stopped or completed.
`.../dose_prescription`	The prescription in dose units (MU or Gy) to convert into charge prescription. When this IO is modified, the `charge_prescription` is automatically updated based on the `dose_to_charge_expression` conversion expression.

IO Path	Description
`.../charge_prescription`	The prescription based on charge units (nC). This IO is only writable when the `dose_prescription` is set to 0.
`.../charge_accumulation`	The accumulated charge during the session.
`.../time_elapsed`	The total elapsed time in seconds over the session since the first start command.
`.../time_active`	The time in seconds during the session that the controller was started and not paused.

Controller States

The `state` IO represents the state of the controller using a short and pain-English string.

State	Description
`ready`	The controller is ready for accepting new prescription settings and can be started.
`dosing`	The controller is active and potentially applying dose to reach the target. The dose may be disabled if the controller is gated by an external signal.
`locked`	The controller is stopped and unable to start due to interlocks being in fault.
`paused`	The controller was paused after starting and is now waiting to be started again or stopped permanently.
`stopped`	The controller was stopped and is waiting to be reset. The stop may have been from a stopping interlock faulting or an external stop command.
`completed`	The controller has successfully completed a dose target and is now waiting for a reset command.

Controller Command Buttons

There are 4 button IO used to command the state of the controller. All the buttons only

Button	Description
`start_button`	Starts the controller if all the interlocks are ok, and the controller is in the `ready` or `paused` state.
`pause_button`	Pauses the controller if it is currently in the `dosing` state. The session can be resumed with another `start_button` command.
`stop_button`	Permanently stops the controller if it is in the `dosing` or `paused` states. After the controller is stopped it cannot be resumed and must be reset to start a new session. This is equivalent to “terminating” a treatment if used in a clinical setting.
`reset_button`	If the controller is in the `ready`, `stopped`, or `completed` state this command will reset all of the accumulated values in the controller and prepares the controller for a new prescription if the interlocks permit it.

If at any time during a session you want to pause or stop the session, the `pause_button` or `stop_button` can be used, and the controller will respond accordingly. The successful result of this command can be confirmed by optionally monitoring the controller state.

Calibration and Configuration IO

These IO are rarely changed programmatically very often but may need to verify or modified by a client.

IO Path	Description
`.../dose_to_charge_expression`	STRING Expression used to convert from dose (MU) to charge (nC).
`.../threshold_current`	NUMBER The minimum amount of input current in nA required to consider dosing to be detected. This will be the threshold used by interlocks that are tolerant to dose drop out.

Q-Pulse Charge Monitor IO

These IO are related to the Q-Pulse charge monitor outputs and configurations.

IO Path	Description
`.../q_pulse_dose`	NUMBER The amount of dose in MU per Q-Pulse. When this IO is modified, the `q_pulse_charge` is automatically updated based on the `dose_to_charge_expression` conversion expression.
`.../q_pulse_charge`	NUMBER The amount of charge in nC per Q-Pulse. This IO is only writable when the `q_pulse_dose` is set to 0.
`.../q_pulse_min_width`	NUMBER The period of time in microseconds for the minimum Q-Pulse width. Q-Pulses may be longer but will never be shorter than the time period.
`.../q_pulse_count`	READONLY NUMBER The number of Q-Pulses that have been outputted so far during the session. Resets to 0 after reset.

Quadrant Detector IO

If a dose controller supports quadrant detectors (typical of any dose controller with 4 or more inputs) the following IO can be used to monitor and control the quadrant detector related functions. Not all dose controllers will support quadrant detectors, so be sure to verify the support either in the user manual for the device or by checking the GUI for the corresponding IO.

IO Path	Description
`.../position_x`	READONLY NUMBER The measured position in mm along the X axis.
`.../position_y`	READONLY NUMBER The measured position in mm along the Y axis.
`.../distance_from_nominal`	READONLY NUMBER The distance in mm from the nominal position.
`.../beam_sigma`	NUMBER The expected beam size sigma in mm. Used to convert into mm units assuming a gaussian shaped beam.
`.../nominal_x`	NUMBER The nominal position in mm along the X axis.
`.../nominal_y`	NUMBER The nominal position in mm along the Y axis.

4.5.2 Controller Interlock and Permits

Ready Permit

This permit must be granted before starting or resuming a session. This permitting being revoked will not cause the controller to pause or stop, but it will prevent a resume or start. The intention of this permit is to control workflow and gate on interlocks that may only be ok while the dose source is off. `.../ready_permit`

Interlock Path	Description
`.../safety_test_pass_interlock`	The safety test must pass before starting or resuming a session.

Stopping Permit

If this permit becomes revoked while running, the controller will automatically stop. `.../stopping_permit`

Interlock Path	Description
`.../elapsed_time_limit_interlock`	The total elapsed time must be less than the given limit.
`.../depositing_current_range_interlock`	The dose rate, while the dose is enabled and detected, must be within a safe range.
`.../enabled_current_range_interlock`	The dose rate, while the dose is enabled, must be within a safe range.
`.../disabled_current_limit_interlock`	The dose rate, while the dose is disabled, must be below the safe limit.
`.../safety_depositing_agreement_interlock`	The safety input and depositing status must agree.
`.../safety_state_interlock`	The safety input must be in the safe state.
`.../distance_from_nominal_interlock`	While the dose is enabled, the beam position must be within the distance limit.
`.../hv_state_interlock`	The high voltage must be in the safe state to deliver dose.

Pausing Permit

If this permit becomes revoked while running, the controller will automatically pause. `.../pausing_permit`

Interlock Path	Description
`.../safety_state_interlock`	The safety input must be in the safe state.
`.../distance_from_nominal_interlock`	While the dose is enabled, the beam position must be within the distance limit.
`.../hv_state_interlock`	The high voltage must be in the safe state to deliver dose.

4.5.3 Simple Charge Prescription Example

Let's say you want to program a single delivery of 100nC of charge. You would:

1. Set `reset_button` to true to reset the controller if it was not reset already.
2. Set `charge_prescription` to `100` (nC).
3. Set `start_button` to true to start the session.
4. Wait for the delivery to complete, you can optionally monitor the `state` or `charge_accumulation` as an accumulated charge.
5. Record the results of the session, if desired.
6. Set `reset_button` to true to reset the controller.

4.6 IGX Admin IO Interface

4.6.1 Overview

The admin component of IGX manages generic system configuration settings. These settings are standard across all IGX devices. These IO will allow you to monitor and configure things like operating mode, versions, clock settings, and more.

IO Path	Description
`/admin/mode`	STRING Sets the operating mode of the device. See the user manual for more information regarding the implications of this mode. Options: “develop”, “product”, “medical”.
`/admin/device_type`	READONLY STRING The name of the type of device. For example, “T1”, “FX4”, “IX256”, “MX1”, and many others.
`/admin/serial`	READONLY STRING The Pyramid serial number for the device as a string. Typically, a number from 0-99999 but may include alphabetical characters in special circumstances.
`/admin/version`	READONLY STRING The IGX version name. “YY.MM.DDHHmm” format. This format is more human friendly but subject to change.
`/admin/version_int`	READONLY NUMBER The IGX version as an integer, useful for using in hashing or version control systems, not subject to change.
`/admin/clock/system_time_int`	READONLY NUMBER The system time in nanoseconds since January 1st, 1970, requires at least 64-bit representation to work.
`/admin/clock/system_time_string`	READONLY STRING The system time as a string. Displayed in a localized format that honors the configured time zone.

IO Path	Description
`/admin/clock/system_time_zone`	The system time zone configuration string. This string must be in the POSIX style time zone format. See: https://en.wikipedia.org/wiki/Tz_database . If left blank, it will default to GMT. 
`/admin/reset_button`	Resets the IGX software without resetting the entire operating system. This is considered a “soft” reset.
`/admin/reboot_button`	Reboots the entire microprocessor including the operating system and IGX. This is considered a “hard” reset.

4.6.2 Permission Policy IO Interface

Typically, programmers will only want to access permission policy IO if they are implementing and external configuration control system. Programmers can leverage the built-in IGX hash and validation system to ease their implementation or write their own systems from scratch.

Preventing unwanted modification and controlling configurations is a critical part of any medical or high-risk industrial control system. The IGX permission design philosophy uses a two-pronged defense.

1. Prevent IO from being externally mutated with systematic interlocks and permits. External systems can still modify the IO they need to operate the system, but nothing more.
2. Ensure changes to IO are immediately visible and easy for external or internal safety systems to react to. Any unexpected changes are easily detectable, de-risking faulty external systems or human error.

In general, the strategy can be summarized in the following steps.

1. The IGX system is tuned and optionally validated to store the hash and then placed into the “medical” operation mode. This prevents accidental modification of protected IO.
2. An external system can either use IGX to store the validated hash or store it itself.
3. The external system monitors the IGX current hash to check for changes against the validated hash and takes the appropriate action as needed.
4. If IGX has a validated hash, it can use integral interlocks to take immediate and independent action if the current hash changes from the validated hash.

Each permission policy uses the same basic IO interface. The hash and hash validation features are completely optional and will only be used depending on your application.

IO Path	Description
`../state`	READONLY STRING The current state of the permission policy. Options: “mutable”, “immutable”, “mutable_valid”, and “immutable_valid”.
`../hash_stale`	READONLY BOOL True if the current hash has gone “stale” and has not yet been re-calculated to reflect the latest IO values.
`../hash_validated`	READONLY BOOL True if the current hash is equal to the validated hash.
`../validate_button`	BUTTON Triggering this button will set the validated hash to the current hash value. The permission policy of this button is the same as its permission policy parent. So, you can only validate the policy when the IO of this policy are mutable.
`../current_hash`	READONLY STRING The current SHA-256 hash of all the policy IO.
`../validated_hash`	READONLY STRING The last validated hash. By default, this string is empty until the validate button is triggered for the first time.

4.7 IGX Interlocks and Permits IO Interface

4.7.1 Overview

IGX is designed to offer a standardized and uniform interface for the interlocks and permit system.

For a detailed explanation of interlocks and permits, please refer to the IGX Interlocks and Permits.

Common Interlock Control IO

IO Path	Description
`.../enabled`	BOOL Enable or disable the interlock.
`.../state`	READONLY STRING The state of the interlock if enabled and non latching
`.../latching`	BOOL If true will prevent the state of the interlock from returning to the 0 state even if the condition is no longer true. This allows the system to “catch” fast conditions that are only available for a single frame, and remember that it happened. The interlock can be reset, but setting enable to false and then true again. The `unlatch()` function can be used to automate this
`.../condition_time`	FLOAT that specifics in seconds the time required for the error condition to be true. If set to a number greater than 0 the condition of the interlock will have to be true for more than that many seconds for the interlock error-state to reflect that. This is sometimes referred to as a “hysteresis” or “timeout”.

InterlockIO Specializations

DigitalInterlockIO

IO Path	Description
`.../fault_polarity`	BOOL The Boolean value that causes a fault, default = False

RangeInterlockIO

IO Path	Description
`.../lower_limit`	The lower limit to cause an error, ignored if null.
`.../upper_limit`	The upper limit to cause an error, ignored if null.
`.../lower_warning`	The lower limit to give a warning, ignored if null.
`.../upper_warning`	The upper limit to give a warning, ignored if null.

ToleranceInterlockIO

IO Path	Description
`.../minimum`	The minimum absolute value that command should have in order to do this check. **This is an important value to set**, because the difference between two values close to zero is almost always out of tolerance. You must have some lower limit on the difference before an percentage makes sense.
`.../tolerance_warning`	The percentage difference between readback and command that causes a warning, ignored if null.
`.../tolerance_limit`	The percentage difference between readback and command that causes an error, ignored if null.

5 IGX Data Acquisition Table API Guide

IGX's Data Acquisition table records samples from one or more ADCs, plus other selected channels, into rows: a waveform, an event window, or a series of readings around a trigger. It runs in one of two modes. A one-shot capture stops when the buffer is full. A circular buffer keeps running and overwrites its oldest rows, and a client can read it as a continuous stream. This guide covers the capture state machine, the full IO interface, and tested WebSocket examples for both modes.

This guide assumes familiarity with the IGX WebSocket Protocol Guide (see page 25), including the `subscribe_table` and `update_table` events used to read table data over the network.

5.1 The Capture State Machine

A Data Acquisition table starts idle, with `armed` and `active` both false. Writing `true` to `collect_button` arms it. Arming resets the table: previous rows are discarded, `data_generation` changes, and the sample counters return to zero. If a start trigger is enabled, `armed` goes high and the table waits for the trigger; otherwise capture begins immediately. Once capture starts, `active` goes high and the counters advance as rows are written. A one-shot capture stops by itself when `sample_count` reaches `buffer_size`. A circular capture keeps going until the stop trigger fires. Either kind stops immediately when `cancel_button` is pressed. When a capture ends, `armed` and `active` both return to false.

Arming is silently ignored if the table is already armed, has no data columns selected, or could not allocate its memory. Writes to locked settings are also ignored without an error. After pressing `collect_button`, confirm that `armed` or `active` goes high, or that the table's `data_generation` changes.

5.2 IO Reference

A device's Data Acquisition table typically lives at a path like `/<device>/data_acquisition/`. All of the IO below live at that path unless otherwise noted. Some are not shown in the default web GUI (either because the underlying node has no `label`, or because it sits in a config node marked hidden from the UI tree), but every one of them is still fully addressable for `get` / `set` / `subscribe` at its normal path.

5.2.1 Status IO

Name	Detail	Type
`armed`	True from arming until the capture ends, including while waiting for a start trigger and while collecting.	Read-only Digital
`active`	True while samples are being written to the table.	Read-only Digital
`sample_count`	Rows currently in the capture window. In circular mode it stops at `buffer_size` once the ring wraps.	Read-only uint32
`total_sample_count`	Samples written in this capture, including rows since overwritten in circular mode. Compare it with the number of samples you have received to detect an overrun.	Read-only uint32
`wrap_count`	Times the circular write position has returned to row 0. Zero until the first overwrite.	Read-only uint32
`write_index`	Next physical row to be written. After a wrap, rows from here onward are older than the rows before it.	Read-only uint32

Name	Detail	Type
`buffer_period`	Time covered by `buffer_size` at the current sample rate.	Read-only Analog (s)
`acquisition_progress`	Fraction of the capture completed, from 0.0 to 1.0, with the standard progress sub-fields. Becomes indeterminate once a circular capture wraps.	Read-only Progress
`acquisition_gated`	True when gating is suppressing sample writes (see `gating_enabled` below).	Read-only Digital
`external_elapsed_time`	Not written over the API. If a component drives it with a non-zero value, it replaces the table's own elapsed time in the time column.	Read-only Analog

`armed`, `active`, `sample_count`, and `buffer_size` are ordinary sibling IO of the table, not part of its table config. Over WebSocket, subscribing to the table with `subscribe_table` will not report these: subscribe to them individually with an ordinary `subscribe` event instead (for example `/fx4/data_acquisition/active/value`). See Capturing a Buffer Once below.

5.2.2 Table Capacity and Memory

`row_capacity`, `row_count`, and `column_count` live under the table's `config` sub-node, which is hidden from the web GUI's navigation tree but addressable like any other IO, for example `.../data_acquisition/config/row_capacity`. The same values also arrive in the table config of `update_table` messages. The three memory IO sit directly under the table.

Name	Detail	Type
`config/row_capacity`	Maximum number of rows the table can currently hold, derived from `max_data_size_bytes` and the current column count.	Read-only uint32
`config/row_count`	Current number of rows stored in the table.	Read-only uint32
`config/column_count`	Total number of active columns, including the time column.	Read-only uint32
`max_data_size_bytes`	Maximum memory reserved for table data. Row capacity is derived from this and the column count. Restricted to developer-level access; not exposed in the standard end-user UI.	uint32 (Bytes)
`memory_allocated`	True once memory has actually been allocated for the table.	Read-only Digital
`allocated_memory_size`	Size of memory actually allocated for the table.	Read-only Analog (Bytes)

5.2.3 Configurable Parameters

Name	Detail	Type
`buffer_size`	Number of rows in the capture window. A one-shot capture stops when `sample_count` reaches it; a circular capture keeps the most recent `buffer_size` rows. Clipped to `config/row_capacity`.	uint32
`circular_buffer_enabled`	Labeled "Circular Buffer" (alias `rolling`). When on, the table does not stop when full: it keeps the most recent `buffer_size` rows and overwrites the oldest. Latched when the table arms. Locked while the table is armed or holds data, and unlocked a moment after a clear, so read the value back to confirm a write.	Digital
`gating_enabled`	Enables gating of the acquisition.	Digital
`gating_expression`	An IGX expression. While gating is enabled, any sample arriving when this expression evaluates non-zero is skipped instead of written to the table.	String (expression)
`reset_counters_on_clear`	When enabled (the default), the running totals of selected counter-type channels are reset whenever the table is cleared or a new capture is armed.	Digital
`standalone_channels`	Labeled "Acquisition Channels" in the UI: the set of channels included as data columns. The path keeps the name `standalone_channels` for configuration compatibility, so address it by that name. Locked while the table holds data.	Symbol multi-select

`buffer_size` and `circular_buffer_enabled` belong to the `prescription` permission policy. The gating settings, `reset_counters_on_clear`, `standalone_channels`, and `clear_button` belong to `protected`, and `max_data_size_bytes` to `develop`. While a policy is locked, writes to its IO are ignored without an error. Check `writable` in the IO's `index.json` over HTTP, or read the value back after writing it.

5.2.4 Start Trigger and Stop Trigger

`start_trigger` and `stop_trigger` are two independent instances of the same edge-trigger IO group, addressable at `.../data_acquisition/start_trigger/` and `.../data_acquisition/stop_trigger/` respectively. A start trigger decides when collection actually begins once the table has been armed; a stop trigger decides when collection ends early, before `buffer_size` would otherwise be reached. Either, both, or neither can be enabled independently.

Name	Detail	Type
`enabled`	Enables this trigger. With the start trigger disabled, collection begins immediately on arming. With the stop trigger disabled, a one-shot capture ends only when `buffer_size` is reached or `cancel_button` is pressed, and a circular capture ends only on `cancel_button`.	Digital

Name	Detail	Type
`source`	The IO path this trigger watches. Any IO on the device can be used, not just a dedicated trigger input.	Symbol select
`mode`	Rising Edge fires the moment `source`'s value crosses from below `threshold` up to at or above it. Falling Edge fires the moment it crosses back down through `threshold`.	Select (rising/ falling)
`threshold`	The value `source` is compared against to detect the edge. Defaults to 1.	Analog

5.2.5 Control Buttons

All four of these are plain `ButtonIO`. As explained in the IGX Button IO Interface (see page 66) guide, writing `true` triggers the action and the device resets the IO back to `false` on its own immediately afterward, so none of the examples in this guide ever need to write `false` back to a button.

Name	Detail	Type
`collect_button`	Arms the table and resets it, discarding the previous rows. Ignored while the table is already armed.	Button
`cancel_button`	Stops an armed or running capture immediately. Rows already captured are kept.	Button
`universal_button`	A single toggle: arms the table if idle, cancels it if armed or active. Meant to drive a custom "Start/Stop Collection" control; it has no label, so it does not appear in the generic auto-generated web GUI.	Button (hidden in UI)
`clear_button`	Labeled "Clear Table." Erases all rows and resets the counters. Locked while the table is armed. Because arming also resets the table, clearing is only needed to empty the table without starting a capture, or to unlock settings that are locked while the table holds data.	Button

5.2.6 Per-ADC Channel Selection

This IO lives on each ADC component registered with the table (for example `.../adc_name/individual_channels_enabled`), not on the Data Acquisition table itself.

Name	Detail	Type
`individual_channels_enabled`	Labeled "Acquire Individual Channels" in the UI. When enabled, every active channel on this ADC is added as a table column automatically, without selecting them individually in `standalone_channels`. Extra computed values the ADC provides (sums, curve fits, and similar) still need to be selected individually in `standalone_channels` regardless of this setting. Locked while the table holds data.	Digital

5.3 Capturing a Buffer Once

The examples below use `/fx4/data_acquisition` on an FX4 and were tested against a live device; the same events work for any device's table at its own path. They build on the WebSocket Protocol Guide (see page 25): subscribe to the table for its rows, subscribe to the capture-state IO as ordinary fields, press `collect_button` with `set`, then poll with `get` until the capture has ended and every row has arrived.

```
import json
import time

import websocket  # pip install websocket-client

ip = "192.168.1.100"
path = "/fx4/data_acquisition"

ws = websocket.create_connection("ws://" + ip + "/", subprotocols=["json.v1"],
timeout=5)

def send(event, data=None):
    ws.send(json.dumps({"event": event, "data": data}))

state = {}          # latest armed / active / sample_count values
rows = {}           # row index -> cells; rows can be re-sent, so key them by index
generation = None   # data_generation the rows in `rows` belong to

def poll():
    # Selecting fields and tables always returns exactly one update,
    # then exactly one update_table.
    global generation
    send("get", {"fields": True, "tables": True})
    for field_path, pairs in json.loads(ws.recv())["data"].items():
        state[field_path.split("/")[-2]] = pairs[-1][0]
    table = json.loads(ws.recv())["data"].get(path)
    if table:
        if table["data_generation"] != generation:
            generation = table["data_generation"]
            rows.clear()
        for index, cells in table.get("rows", []):
            rows[index] = cells

# Capture state is not part of the table, so subscribe to it as ordinary fields.
send("subscribe", {path + "/" + name + "/value": False for name in ("armed", "active",
"sample_count")})
send("subscribe_table", {path: True})
poll()
start_generation = generation

# Arming resets the table and starts a new data generation; no separate clear is
needed.
send("set", {path + "/collect_button/value": True})

deadline = time.monotonic() + 30.0
while time.monotonic() < deadline:
    poll()
    finished = generation != start_generation and not state["armed"] and not
 state["active"]
    # Rows are published about every 100 ms, so the last ones can arrive
```

```
    # after active goes false. Keep polling until all of them are in.
    if finished and len(rows) >= state["sample_count"]:
        break
    time.sleep(0.05)
else:
    raise TimeoutError("capture did not finish; check that the table armed")

data = [rows[i] for i in range(len(rows))]
print("Captured", len(data), "rows; first row:", data[0])
ws.close()
```

A few notes on this pattern:

- Each `get` selects both fields and tables, so the device always sends exactly one `update` and then exactly one `update_table`. Reading both keeps requests and replies in step; reading only one message per `get` would let unread replies pile up on the connection.
- Rows arrive as `[row_index, cells]` and can be re-sent, so store them by index rather than appending. When `data_generation` changes, the table was reset: discard rows from the previous generation.
- The device publishes one-shot rows about every 100 ms and limits how many rows go into each message, so `active` can go false before the last rows arrive. A capture is complete only when `armed` and `active` are both false and you hold `sample_count` rows. In testing on an FX4, a 4000-row capture arrived complete and in order.
- With a start trigger enabled, row 0 holds a snapshot taken just before the trigger.

5.4 Continuous Acquisition with a Circular Buffer

To stream without gaps, turn on `circular_buffer_enabled`. The table then keeps running after it fills: it holds the most recent `buffer_size` rows and overwrites the oldest one each time a new sample arrives. In `update_table`, `ring_capacity` equals `buffer_size` during a circular capture, and row indices are physical positions in the ring, so sample *n* of the capture is stored in row *n* % `ring_capacity`. A client that polls before the ring wraps past its unread rows receives every sample exactly once.

The example below enables the circular buffer, streams for 60 seconds, then cancels the capture and drains the remaining rows. In testing on an FX4 sampling at 2 kHz, with a 4000-row ring and a 50 ms poll interval, it reconstructed more than 12,000 consecutive samples in 6 seconds with no gaps or duplicates.

```
import json
import time

import websocket  # pip install websocket-client

ip = "192.168.1.100"
path = "/fx4/data_acquisition"
buffer_size = 4000  # rows of history the device keeps; size it to cover several polls

ws = websocket.create_connection("ws://" + ip + "/", subprotocols=["json.v1"],
timeout=5)

def send(event, data=None):
    ws.send(json.dumps({"event": event, "data": data}))

def handle_rows(new_rows):
    pass  # append to a file, plot, etc.

state = {}
generation = None
```

```
next_sample = 0  # samples received so far in this capture

def poll():
    global generation, next_sample
    send("get", {"fields": True, "tables": True})
    for field_path, pairs in json.loads(ws.recv())["data"].items():
        state[field_path.split("/")[-2]] = pairs[-1][0]
    table = json.loads(ws.recv())["data"].get(path)
    if not table:
        return
    if table["data_generation"] != generation:
        generation = table["data_generation"]
        next_sample = 0
    ring = table["ring_capacity"]  # buffer_size during a circular capture, 0
otherwise
    new_rows = []
    for index, cells in table.get("rows", []):
        # Accept only the physical row that holds the next sample; skip re-sent rows.
        if index == (next_sample % ring if ring else next_sample):
            new_rows.append(cells)
            next_sample += 1
    if new_rows:
        handle_rows(new_rows)

names = ("armed", "active", "total_sample_count", "circular_buffer_enabled")
send("subscribe", {path + "/" + name + "/value": False for name in names})
send("subscribe_table", {path: True})

# Circular Buffer is locked while the table is armed or holds data, and it
# unlocks a moment after a clear. Write it, read it back, and retry.
send("set", {path + "/clear_button/value": True})
for _ in range(40):
    send("set", {path + "/buffer_size/value": buffer_size, path + "/
circular_buffer_enabled/value": True})
    poll()
    if state.get("circular_buffer_enabled"):
        break
    time.sleep(0.05)
else:
    raise RuntimeError("Circular Buffer stayed off; check the table is idle and the
permission policy")

send("set", {path + "/collect_button/value": True})

stop_at = time.monotonic() + 60.0
while time.monotonic() < stop_at:
    poll()
    if state.get("total_sample_count", 0) - next_sample > buffer_size:
        print("Overrun: unread rows were overwritten; poll faster or raise
buffer_size")
    time.sleep(0.05)

send("set", {path + "/cancel_button/value": True})

# Drain the rows written before the cancel took effect.
for _ in range(20):
    poll()
    if not state["active"] and next_sample >= state["total_sample_count"]:
```

```
            break
        time.sleep(0.05)

print("Streamed", next_sample, "samples")
ws.close()
```

5.4.1 Practical Considerations

- **Size the ring for your poll interval.** The ring must hold more samples than arrive between polls, with margin for network delays. Rows overwritten before you read them are lost. If `total_sample_count` minus the samples you have received exceeds `buffer_size`, an overrun has occurred.
- **Confirm the mode before arming.** `circular_buffer_enabled` is locked while the table is armed or holds data, and it unlocks a moment after a clear. Write it, read it back, and retry, as the example does. A write that is ignored leaves you with a one-shot capture.
- **Restore shared settings.** `buffer_size` and `circular_buffer_enabled` are saved settings that persist after the client disconnects. If other users expect one-shot captures, clear the table, then set them back and read them back.
- **Rows from several ADCs.** When ADCs share a sample clock, a row can be re-sent after a slower ADC fills in its columns, and the first copy may contain `null` cells. The example keeps the first copy. If that matters, hold recent rows back briefly, or keep updating stored rows by index.
- **Repeated one-shot captures** are an alternative when gaps between buffers are acceptable. Wait until every row of a capture has arrived before pressing `collect_button` again, because arming resets the table and discards rows not yet delivered.

6 IGX File Format Specifications

6.1 IGX JSON IO Files

6.1.1 Introduction

All IGX data is stored in objects called "Fields", which are stored in objects called "Nodes". Nodes can contain Fields and other Nodes. The top-level Node is a special Node called the "root", which is the only Node without a parent.

Both Nodes and Fields use a string to represent their name, which identifies the Node or Field within the parent Node. This entire structure is referred to as the "vertex tree" or "vtree" for short. The structure of this tree mimics what you would find in a filesystem, and like a filesystem, you can refer to each Node or Field by its "path", which is the concatenation of that vertex name and the names of all of its ancestor Nodes. These paths are how IGX will uniquely identify any particular object in the vtree.

Example JSON

The JSON is structured using objects to represent nodes and key/value pairs to represent fields. For example:

```
{
  "field_a": "value a",
  "field_b": 1.234,
  "child_a": {
    "field_a": "value aa",
    "field_b": 432.1,
    "child_aa": {
      "field_c": false,
      "field_d": [1.234, 5.678]
    }
  },
  "child_b": {
    "field_e": [[1.234, 3.456], [5.532, 32.33]]
  }
}
```

The path of `field_d` within `child_aa` would be `/child_a/child_aa/field_d`.

Field Types

Fields in IGX are standardized but can be expanded as the project develops. Currently, there are a handful of standard fields you can find in the JSON today.

Field	Type	Required	Notes
name	string	Yes	Name of the Node.
type	string	Yes	Name of the C++ class used by this Node.
label	string	No	Human-friendly name for the Node, used by GUIs.
detail	string	No	Brief description of the Node.
hidden	bool	No	`True` if this Node is too complicated for the average user.
color	string	No	Name of a color to assign to this Node.
icon	string	No	Name of an icon to use for this Node.

Field	Type	Required	Notes
value	any	No	The value for this Node, if it is an I/O type.
readonly	bool	No	`True` if this Node's value should only be read and not written.
units	string	No	The unit of measure for the value field (e.g., "V", "A", "mA", etc.).
format	string	No	The formatting specifier of how to format the value number.

Index JSON Files

Every Node on the vertex tree has a corresponding `index.json` file under its path in the `/io` directory. This file contains the JSON for this Node and all child Nodes. So, if you read the `/io/index.json` file, you will be reading a file containing all the fields and nodes for the entire tree. If you read the `/io/child/child/index.json` file, you will only get the fields and nodes for the `/child/child` node.

Field JSON Files

In addition to the `index.json` files, there are also JSON files for every individual field on the vertex tree that contain only the value of that field. For example, if we had a `/io/index.json` file that looked like this:

```
{
  "field_a": 1.234,
  "child_a": {
    "field_b": [true, true, false]
  }
}
```

The body of the `/io/field_a.json` file would be:

```
1.234
```

The body of the `/io/child_a/field_b.json` file would be:

```
[true, true, false]
```

These field files provide an easy way to grab only the data you care about, although if you plan on querying many fields, it will almost always be best to grab an `index.json` file instead and parse out what you need.

JSON Schema

This JSON schema represents the structure of IGX data, which consists of a hierarchical arrangement of "Nodes" and "Fields". Nodes can contain other nodes or fields as key/value pairs. The schema enforces that each node must have a "name" and a "type", and also allows for optional properties such as "label", "detail", "hidden", "color", "icon", "value", "readonly", "units", and "format". The schema ensures that the IGX data adheres to the defined structure and properties.

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "IGX Index Files",
  "description": "IGX schema for nodes index files",
  "type": "object",
  "patternProperties": {
    "^[a-zA-Z0-9_]+$": {
```

```
      "oneOf": [
        {
          "type": "object",
          "properties": {
            "name": { "type": "string" },
            "type": { "type": "string" },
            "label": { "type": "string" },
            "detail": { "type": "string" },
            "hidden": { "type": "boolean" },
            "color": { "type": "string" },
            "icon": { "type": "string" },
            "value": { "type": ["string", "number", "boolean", "array", "object"] },
            "readonly": { "type": "boolean" },
            "units": { "type": "string" },
            "format": { "type": "string" }
          },
          "required": ["name", "type"],
          "additionalProperties": false
        },
        { "$ref": "#" }
      ]
    }
  },
  "additionalProperties": false
}
```

Availability of Protocols

The `index.json` and `field.json` files are special text files that exist on the server's file system. As such, they can be read and written exactly like any other normal file. You may use SFTP, HTTP, SSH, or, if your target machine is running QNX, Qnet[48]. You can use libraries to directly connect using one of these protocols or mount the server filesystem to your local machine and use standard file I/O to interact with IGX.

See the IGX Network Protocols (see page 10) section for a full overview of all available protocols.

Support for new protocols can be added as needed, so if you don't see a protocol you'd like to use, let the Pyramid's staff know.

Helpful Links

- JSON Specification[49]
- SFTP - Secure File Transfer Protocol[50]
- HTTP - Hypertext Transfer Protocol[51]
- SSH - Secure Shell[52]
- Qnet - QNX Networking Protocol[53]

48. https://www.qnx.com/developers/docs/7.1/#com.qnx.doc.neutrino.sys_arch/topic/qnet.html
49. https://www.json.org/
50. https://en.wikipedia.org/wiki/SSH_File_Transfer_Protocol
51. https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol
52. https://en.wikipedia.org/wiki/Secure_Shell
53. https://www.qnx.com/developers/docs/6.5.0_sp1/index.jsp?topic=%2Fcom.qnx.doc.neutrino_prog%2Fqnet.html