



PYRAMID
www.pyramid.tech

IGX

IGX - Programmer Manual

Document ID: 4140138503

Version: v7

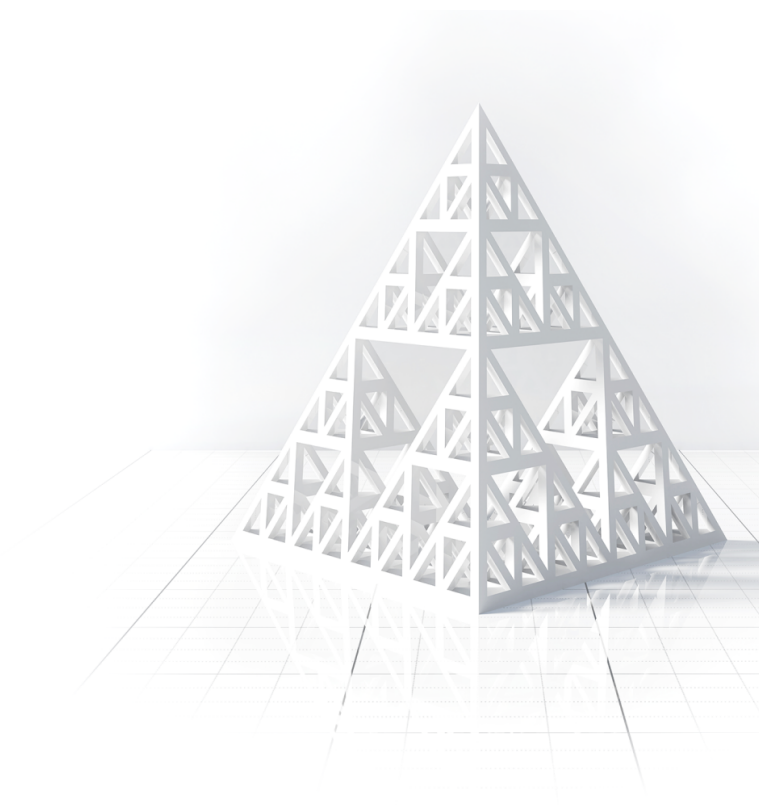


Table of Contents

1	Introduction.....	7
2	Document Control	8
2.1	Version History	8
2.2	Approvals	8
2.2.1	Signatures	8
3	IGX Network Protocols.....	9
3.1	Overview of IGX Protocols	9
3.1.1	HTTP Protocol.....	9
3.1.2	WebSocket Protocol.....	9
3.1.3	EPICS Protocol	10
3.1.4	SFTP Protocol	10
3.1.5	Qnet Protocol	10
3.2	IGX HTTP Protocol Guide	11
3.2.1	IGX HTTP Server Introduction	11
3.2.2	HTTP Quick Start	11
	Use cURL to GET field value.....	11
	use cURL to PUT a field value.	11
	Use Microsoft Excel to GET field value.	12
	Use Python and requests library to SET and GET field value.	12
	Suitable HTTP Libraries	12
3.2.3	The HTTP Protocol Basics	13
	The TCP Connection	13
	HTTP URLs	13
	HTTP Request Structure	14
	HTTP Response Structure	14
3.2.4	HTTP Python Examples.....	15
	Python Requests Sessions	15
	Sending a GET Request	15
	Sending a PUT Request	16
3.2.5	Postman for HTTP Testing	17
	Sending a GET Request	17
	Sending a PUT Request	18
3.2.6	PLC HTTP Programming	18
	Converting Strings to Numeric or Binary Values	19
	Protocol Gateways	19

Custom Code Solutions	19
3.3 IGX WebSocket Protocol Guide	21
3.3.1 WebSocket Overview	21
3.3.2 IGX JSON Message Protocol	23
Config Event.....	23
Subscribe Event.....	25
Subscribe Table Event	26
Get Event	27
Set Event	27
Update Event	28
Update Table Event.....	29
Update ID Event.....	30
Get ID Event	31
Typical Protocol Sequence	32
3.3.3 Suitable WebSocket Libraries	33
3.3.4 Python T1 Example	34
3.3.5 Conclusion	36
3.4 IGX Apache Proxy Server Guide	36
3.4.1 Prerequisites	37
3.4.2 Setup Notes.....	37
3.4.3 Steps to Configure Apache Proxy.....	37
1. Edit the Apache Configuration File.....	37
2. Enable Required Apache Modules	38
3. Verify Configuration	38
3.4.4 Explanation of Configuration	38
ProxyPass and ProxyPassReverse.....	38
Dynamic Mapping with RewriteRule	38
3.5 IGX EPICS Protocol Guide	38
3.5.1 EPICS Overview	38
EPICS Network Protocols and Ports	39
Automatic Network Discovery.....	40
Using EPICS Across Network Barriers.....	41
EPICS CA Address Configuration.....	41
Choosing EPICS for Your Control System.....	42
3.5.2 IGX EPICS Interface	42
Handling Multiple IGX Devices	43
IGX PV Alias Map File	43
3.5.3 EPICS Utility Programs	44

Getting the Current Hostname	45
Setting a New Hostname	45
Monitoring Heartbeat	45
3.5.4 EPICS Python Examples	46
Python Library Overview.....	46
Getting a PV (IO) Value	47
Putting a PV (IO) Value.....	47
Zeroing a T1 Probe	48
Polling Heartbeat	49
Subscribing to Heartbeat.....	50
3.6 IGX SFTP Protocol Guide.....	51
3.6.1 Use SFTP for IGX IO Data	51
Accessing IGX IO Data via SFTP	51
Python Example	51
3.7 IGX Qnet Protocol Guide	52
3.7.1 Qnet Overview	52
Network Protocol.....	52
Device Discovery	53
The Net Directory	53
Network Packet Passing.....	53
File Exposure between Nodes.....	54
3.7.2 Integrating Qnet with IGX.....	54
Accessing IO Data through Qnet.....	54
File Path Conventions in IGX	54
3.7.3 Python Qnet Example.....	54
4 IGX Standard IO Interfaces.....	56
4.1 IGX Button IO Interface.....	56
4.1.1 Overview.....	56
4.1.2 Button Variants	56
4.1.3 Triggering a Button.....	56
4.1.4 Python HTTP Example.....	57
4.1.5 Hidden "Spinning" Indicator.....	57
4.1.6 Real-World Example: Data Acquisition Table Controls	57
4.2 IGX High Voltage IO Interface	58
4.2.1 Overview.....	58
Interface IO.....	58
Module States	58
Safety Interlocks.....	58

Internal Voltage Sense Circuit	59
External Voltage Sense Circuit	59
4.2.2 Use Case Examples	59
Setting and Enabling High Voltage Output	59
Reading Internal Voltage Values	59
Monitoring External Voltage Values (Optional)	59
4.3 IGX Relay IO Interface	60
4.3.1 Basic Relay Control IO	60
4.3.2 Advanced Control IO	60
4.3.3 Web GUI	60
4.4 IGX Offset Correction (Zero) IO Interface	61
4.4.1 Overview	61
Interface IO	61
4.5 IGX Dose Controller IO Interface	62
4.5.1 Overview	62
Primary Control IO	62
Calibration and Configuration IO	64
Q-Pulse Charge Monitor IO	64
Quadrant Detector IO	64
4.5.2 Controller Interlock and Permits	65
Ready Permit	65
Stopping Permit	65
Pausing Permit	65
4.5.3 Simple Charge Prescription Example	66
4.6 IGX Admin IO Interface	66
4.6.1 Overview	66
4.6.2 Permission Policy IO Interface	67
4.7 IGX Interlocks and Permits IO Interface	68
4.7.1 Overview	68
Common Interlock Control IO	68
InterlockIO Specializations	68
5 IGX Data Acquisition Table API Guide	70
5.1 The Capture State Machine	70
5.2 IO Reference	70
5.2.1 Status IO	70
5.2.2 Table Capacity and Memory	71
5.2.3 Configurable Parameters	71

- 5.2.4 Start Trigger and Stop Trigger 72
- 5.2.5 Control Buttons 72
- 5.2.6 Per-ADC Channel Selection..... 72
- 5.3 Capturing a Buffer Once..... 73
- 5.4 Pseudo-Continuous Acquisition 74
 - 5.4.1 Practical Considerations 74
- 6 IGX File Format Specifications 76**
 - 6.1 IGX JSON IO Files 76
 - 6.1.1 Introduction 76
 - Example JSON 76
 - Field Types 76
 - Index JSON Files..... 77
 - Field JSON Files 77
 - JSON Schema 77
 - Availability of Protocols..... 78
 - Helpful Links..... 78

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
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		Harvey Jules Nett	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

Tuesday, Jul 21, 2026, 04:47 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.

3.1.1 HTTP Protocol

All IGX devices are equipped with a high-performance HTTP file server that allows you to build external applications with ease. This file server can read and write files on the device's file system, granting access to all files on the device.

IGX utilizes plain text JSON files to store I/O data, which can be queried and modified periodically to collect or update dynamic data effortlessly. These dynamic files reside in a special directory at the root of the file system called `/io`. To access the I/O data through HTTP, prepend `/io` to the beginning of the I/O path.

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.
- Lacks support for lossless buffered data.

See the full guide here: [IGX HTTP Protocol Guide \(see page 11\)](#)

3.1.2 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.

In the context of IGX, WebSocket provides a convenient and efficient method for exchanging data between a client (e.g., a Python script) and an IGX device. This protocol is particularly suited for scenarios that require continuous, real-time updates, such as monitoring sensor values or controlling actuators.

Strengths

- Enables streaming lossless buffered high-speed data.
- Can manage hundreds of IO through a single TCP connection.
- 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.

See the full guide here: [IGX WebSocket Protocol Guide \(see page 21\)](#)

3.1.3 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.

With IGX devices, you can leverage the power of EPICS to integrate them with existing control systems or build custom applications for scientific and industrial purposes. This ensures smooth interoperability with a wide range of hardware and software components in your infrastructure.

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 \(see page 38\)](#)

3.1.4 SFTP Protocol

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

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.

See the full guide here: [IGX SFTP Protocol Guide \(see page 51\)](#)

3.1.5 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. By integrating Qnet, you can simplify the interaction between various nodes in the system, eliminating the need for specialized APIs or custom communication protocols. Qnet's transparent networking capabilities ensure efficient management and exposure of IO data through a virtual file system, facilitating easy access and manipulation of data between devices.

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 \(see page 52\)](#)

3.2 IGX HTTP Protocol Guide

3.2.1 IGX HTTP Server Introduction

All IGX devices come with a high-performance [HTTP](#)¹ file server that can be used to build external applications. This file server is capable of reading and writing files on the device's file system. All files on the device can be queried through the HTTP server.

IGX uses plain text JSON files to store IO data, which can be queried and written periodically to collect or set dynamic data easily. These dynamic files are stored in a special directory located at the root of the file system called `/io`. When accessing the IO data through HTTP, you will need to prepend `/io` to the beginning of the IO path.

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.

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
```

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

2. <https://en.wikipedia.org/wiki/JSON>

3. <https://curl.se/>

4. <https://curl.se/>

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 requests
import json

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

print(requests.put(url, json.dumps("NEW-NAME")).json())
print(requests.get(url).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.
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.

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/>

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/elnoormous/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>

Language	Library	Notes
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.

It's worth noting that IGX allows a single TCP socket to be reused multiple times for sending and receiving multiple messages. All TCP messages are plain text with ASCII encoding. All IGX responses are in plain text too, with the bodies encoded using the JSON format.

TCP Keep Alive

If you are sending multiple requests in quick succession or very frequently, it helps to reuse the same TCP connection for all those requests. The [requests](#)²⁰ library for Python does this automatically if you use a session object (see example later), so you don't have to worry about it. However, if you're using a different library or language, make sure that it keeps the connection alive between requests to improve performance.

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.

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

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

- **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 currently supports two verbs: `GET`, which is used to request a file, and `PUT`, which is used to write files. Below is an example of the simplest possible request:

```
GET /io/heartbeat/value.json HTTP/1.1
```

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 \(MDN\) HTTP²¹ Messages](https://developer.mozilla.org/en-US/docs/Web/HTTP/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: 1
Content-Type: application/json
Content-Encoding: identity
Content-Length: 4

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

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

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 URL requested returns a Boolean value, so only the value is returned in the response body. The response body contains the actual response data, which in this example is a Boolean value of `true`.

The response body contains the actual response data, which in this example is a JSON-encoded value or object depending on the resource requested. The value of the data field contains the value of the requested IO field.

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: Request was successful.
- 400 Bad Request: Request was malformed or invalid.
- 404 Not Found: Requested resource was not found.
- 500 Internal Server Error: Server encountered an error while processing the request.

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 object or creating a session object and making a request from that. Sessions are usually the superior choice for production code, but it does complicate the examples. If you want to make requests to multiple devices, you will need to create a session object for each device.

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

Sending a PUT request will modify the file or IO on the device. If the IO or file is read-only, the operation will fail. The structure of the PUT request is just like the GET request, except this time the client supplies the body (the value).

The following is an example of the new request structure:

```
PUT /io/net/hostname.json HTTP/1.1

"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

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

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"

def putIOValue(path, value):
    return requests.put("http://" + ip + "/io" + path + "/value.json",
                        json.dumps(value), timeout=1.0).json()

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 putHostname(value):
    return getIOValue("/net/hostname", value)

print("Old Hostname", getHostname())
putHostname("New-Hostname")
print("New Hostname", getHostname())

```

In this example, the `putIOValue` function is used to send a PUT request to IGX to set the value of an IO field. The `getIOValue` function is used to send a GET request to IGX to retrieve the value of an IO field. The `getHostname` and `putHostname` functions use `getIOValue` and `putIOValue`, respectively, to get and set the value of the hostname IO field.

3.2.5 Postman for HTTP Testing

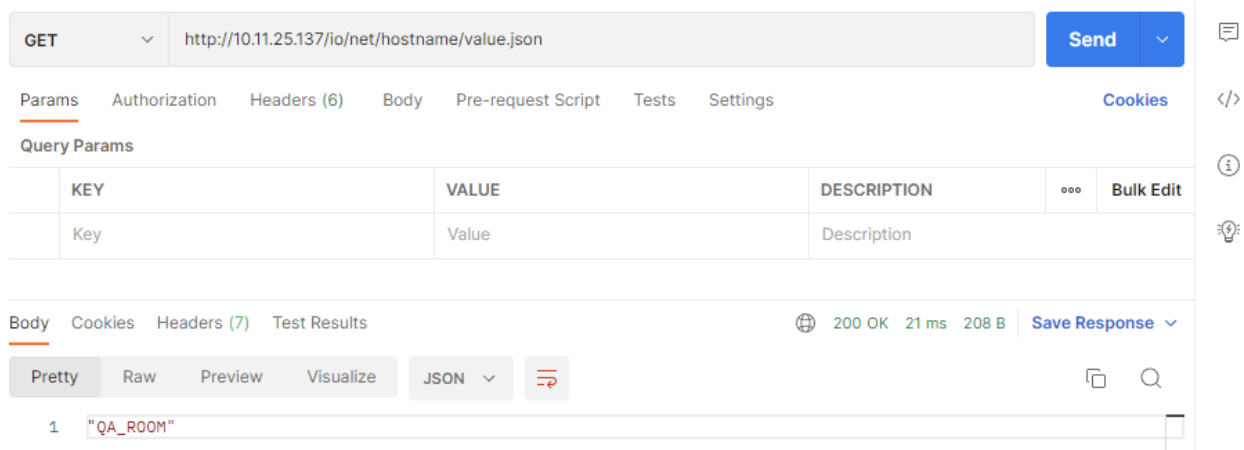
Postman is a powerful tool for sending and receiving HTTP messages with a user-friendly GUI interface. It can be invaluable for debugging your code and testing new queries without the hassle of programming. You can download Postman from the following link:

<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.

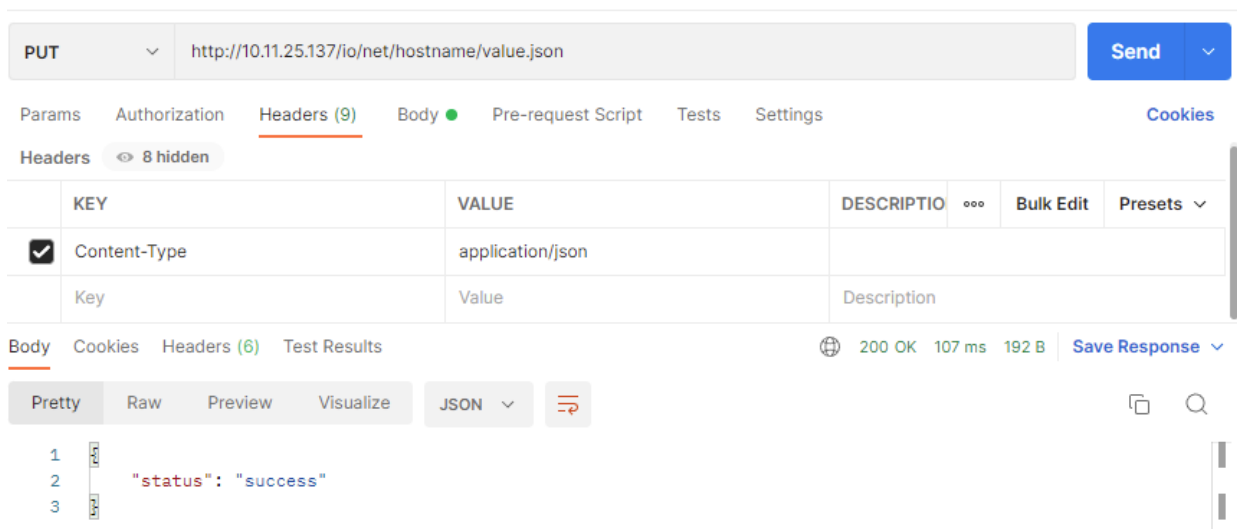


3. Click the Send button and observe the response.

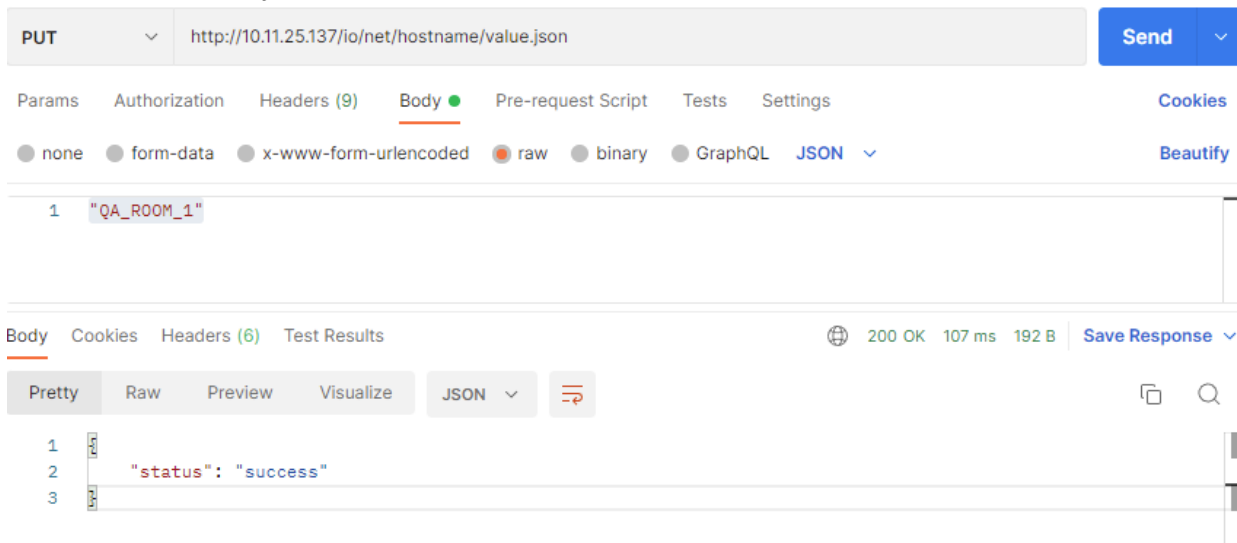
Sending a PUT Request

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. Add the header `Content-Type: application/json`. This tells IGX that the content you are sending is encoded using the JSON format.



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.

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>

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>
#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 is a communication protocol that enables two-way, real-time communication between a client (frequently a web browser) and a server over a single, long-lived connection. It is designed to work over the same ports as HTTP to allow for easy integration with existing web infrastructure.

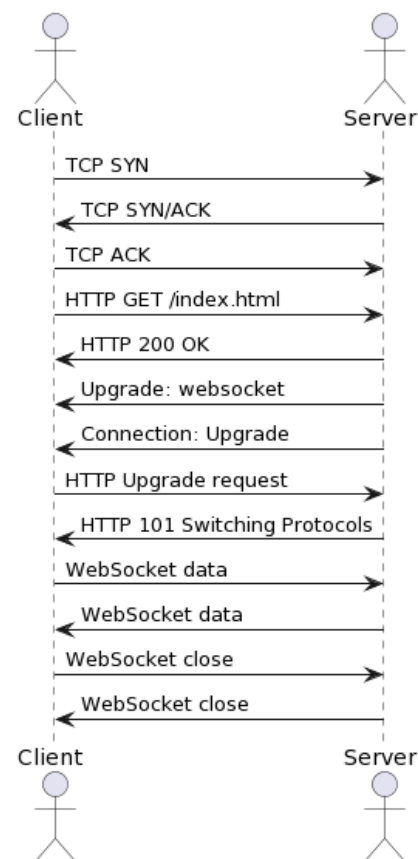
In the context of IGX, the WebSocket protocol provides a convenient and efficient way to exchange data between a client (such as a Python script) and an IGX device. The protocol is particularly well-suited for use cases where continuous, real-time updates are required, such as monitoring sensor values or controlling actuators.

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.

TCP, HTTP, and WebSocket Sequence Diagram



1: WebSocket Timing Diagram

Connection Establishment: The client initiates a WebSocket connection to the IGX device by sending an HTTP request with an "Upgrade" header indicating a desire to switch to the WebSocket protocol. The device, upon accepting the request, sends an HTTP response with a corresponding "Upgrade" header, and the connection is then switched from HTTP to WebSocket. The device always responds with a `Sec-WebSocket-Protocol` header naming the protocol it selected (`json.v1` by default), even if the client didn't request one. Some clients, including web browsers, treat an unrequested `Sec-WebSocket-Protocol` response as a handshake failure, so it's safest to always advertise a subprotocol explicitly, the same way IGX's own web client does:

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

The device also accepts `mpack.v1` in place of `json.v1`, which switches the same event/data protocol to a MessagePack binary encoding instead of JSON text. This guide covers the JSON encoding throughout; the message structure and event names are identical either way, only the wire encoding changes.

Message Exchange: Once the WebSocket connection is established, the client and the device can exchange messages in a bidirectional, real-time manner. In the case of IGX, the messages are formatted as JSON objects containing "event" and "data" fields described in more detail later.

```
ws.send(json.dumps({"event": event, "data": data}))
response = json.loads(ws.recv())
```

Message Handling: Both the client and the device should implement appropriate handlers to process incoming messages based on their "event" type. In the provided Python example, the

`onMessageEvent(event, data)` function handles incoming "update" events from the device, processes the received data, and stores it in a local database. This handler is described in more detail later.

```
def onMessageEvent(event, data):
    ...
```

Connection Termination: When communication is no longer required, either the client or the device can close the WebSocket connection by sending a close frame and closing the underlying TCP connection. In the provided Python example, the WebSocket connection is closed using the `ws.close()` function.

```
ws.close()
```

By leveraging the WebSocket protocol, IGX devices can efficiently communicate with clients in real-time, allowing for continuous monitoring and control of various parameters, such as sensor values or actuator states. The protocol is well-suited for use cases that require low latency and minimal overhead, making it an ideal choice for many industrial and IoT applications.

3.3.2 IGX JSON Message Protocol

The messages are structured as JSON objects containing two main fields: "event" and "data". The "event" field describes the type of event, while the "data" field contains any additional information associated with the event.

Most events carry ordinary IO field paths (single values, such as `/t1/probe/field/value`), always ending in `/value` and never prefixed with `/io` (that prefix is specific to the HTTP file-server interface, described in the [IGX HTTP Protocol Guide \(see page 11\)](#)). A separate set of events, `subscribe_table` and `update_table`, exists for working with **Tables**, which are grids of rows and columns (for example a captured data acquisition buffer); tables are addressed by their bare vertex path, with no `/io` prefix and no `/value` suffix, since the path identifies the whole table rather than a single value. The `config`, `get`, `set`, `get_id`, and `update_id` events described below apply uniformly to both fields and tables; only subscribing to and receiving data from a table uses its own event names, covered in the [Subscribe Table Event](#) and [Update Table Event](#) sections.

Config Event

This completely optional event can be sent by a client to the server at the start of a session to configure specific parameters about this session. The default session configuration is reasonable for most use cases, but in some cases, you may find these settings helpful to change.

Example Message

```
{
  "event": "config",
  "data": {
    "use_short_id": false,
    "always_update": false,
    "partial_row_update": true,
    "include_hidden_columns": true,
    "init_all_rows": true,
    "update_arrays_as_blobs": false
  }
}
```

JSON Schema

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Config Event",
  "description": "Schema for config event in IGX JSON Message Protocol",
}
```

```

    "type": "object",
    "properties": {
      "event": {
        "type": "string",
        "const": "config"
      },
      "data": {
        "type": "object",
        "patternProperties": {
          "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 },
          "init_all_rows": { "type": "boolean", "default": true },
          "update_arrays_as_blobs": { "type": "boolean", "default": false }
        },
        "additionalProperties": false
      }
    },
    "required": ["event", "data"],
    "additionalProperties": false
  }
}

```

Configurable Parameters

Parameter	Description
<code>use_short_id</code>	False by default. If set to true, the IGX server will use short IDs in its update message events. See the <code>update_id</code> and <code>get_id</code> messages bellow for more information. This applies to table identifiers exactly the same way it applies to field identifiers.
<code>always_update</code>	False by default. If set to true, the IGX server will always include all subscribed data in the update message, regardless of if the data has changed or not. By default, the protocol only sends new data if the data has changed, but this parameter allows you to always get updates regardless. Use caution with this parameter, as it can be easy to create very large update messages if subscribing to many values, which may impact your client performance.
<code>partial_row_update</code>	True by default. Only relevant to table subscriptions. A single table row commit made by the device can occasionally span more rows than can be sent in one internal chunk, in which case it is marked "partial". When true, those partial row ranges are still forwarded to the client as they arrive; when false, the server withholds a partial range until it is completed. Most clients should leave this at its default.
<code>include_hidden_columns</code>	True by default. Only relevant to table subscriptions. Controls whether a table's hidden columns are included in its <code>config</code> and <code>rows</code> data. Set to false to reduce message size if you only care about the visible columns.
<code>init_all_rows</code>	True by default. Only relevant to table subscriptions. Controls whether every existing row is queued up when a table is first subscribed, so the first <code>update_table</code> event delivers a full snapshot. Set to false if the client already has the table's row state from elsewhere and only wants incremental changes going forward.

Parameter	Description
<code>update_arrays_as_blobs</code>	False by default. MessagePack (<code>mpack.v1</code>) sessions only; has no effect over the JSON encoding. When true, numeric float/double arrays are encoded as a compact binary blob instead of an array of tagged numbers, reducing message size. Only enable this if your client explicitly supports decoding the blob form.

Subscribe Event

Sent by the client to subscribe to specific data paths on the device. The `data` field contains a dictionary with the paths as keys and Boolean values indicating whether the data should be buffered or not.

Buffered data will include all the data in an array since the last get event, while unbuffered data will only contain the latest data point.

This message only tells the server that the client intends on using this data. In order to request the data itself, the client must send a get event message.

Each `subscribe` event replaces the client's entire field subscription list with whatever paths are included in that message. There is no separate message for removing a subscription: to stop receiving a path, just send another `subscribe` event that leaves that path out. To subscribe to additional paths without dropping existing ones, include the full set you want subscribed (previous paths plus the new ones) in each `subscribe` event.

This event is for ordinary single-value fields only. To subscribe to a Table, use the Subscribe Table Event below instead.

Example Message

```
{
  "event": "subscribe",
  "data": {
    "/t1/probe/field/value": true
  }
}
```

JSON Schema

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Subscribe Event",
  "description": "Schema for subscribe event in IGX JSON Message Protocol",
  "type": "object",
  "properties": {
    "event": {
      "type": "string",
      "const": "subscribe"
    },
    "data": {
      "type": "object",
      "patternProperties": {
        "^/.*$": { "type": "boolean" }
      },
      "additionalProperties": false
    }
  },
  "required": ["event", "data"],
  "additionalProperties": false
}
```

```
}
```

Subscribe Table Event

Sent by the client to subscribe to one or more Tables on the device, identified by their bare vertex path (no `/io` prefix, no `/value` suffix). As with the field `subscribe` event, sending this message replaces the entire table subscription list; it does not add to whatever was subscribed before. The boolean value in the `data` object is currently unused by the server; include `true` for each path for forward compatibility.

There is likewise no separate message for removing a table subscription. Send another `subscribe_table` event that leaves out the table you want to stop receiving, and, as above, include every table you still want subscribed in that same message.

Subscribing to a table does not by itself send any data. The client must still send a `get` event afterward. The first `update_table` event received after subscribing to a table always contains a full snapshot of every row currently stored in that table (unless `init_all_rows` was set to false via the Config Event above); subsequent updates contain only new or changed rows.

- 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 \(see page 70\)](#) for a full worked example.

Example Message

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

JSON Schema

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

Get Event

Sent by the client to request new data from the device since the last `get` event. If the client has already sent a `subscribe` and/or `subscribe_table` event, no additional data field is required for this message. After receiving a get event, the IGX service will send an `update` event (for any subscribed fields with new data) and/or an `update_table` event (for any subscribed tables with new data).

Clients should send a new get event whenever they want to receive another update message. Clients should not send more than one `get` event without first receiving a response. This is to prevent the server from being overloaded with incoming messages.

Example Message

```
{
  "event": "get"
}
```

JSON Schema

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Get Event",
  "description": "Schema for get event in IGX JSON Message Protocol",
  "type": "object",
  "properties": {
    "event": {
      "type": "string",
      "const": "get"
    }
  },
  "required": ["event"],
  "additionalProperties": false
}
```

Set Event

Sent by the client to request a modification to a field value. The `data` field is a dictionary containing all fields and their corresponding new values, exactly mirroring the `subscribe` event's path format but without the boolean. It is wise to subscribe to fields you plan on modifying, so you can confirm the value changed in a later update message.

Everything writable in IGX, including buttons and editable table columns, is just a field, so there's no separate message type for triggering an action or writing table data. Setting a [Button IO \(see page 56\)](#)'s path to `true` triggers it the same way setting any other field writes a new value. See the [IGX Data Acquisition Table Guide \(see page 70\)](#) for a worked example of using `set` with table buttons and columns specifically.

Example Message

```
{
  "event": "set",
  "data": {
    "/t1/probe/offset/value": 1.234,
    ...
  }
}
```

JSON Schema

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
```

```

    "title": "Set Event",
    "description": "Schema for set event in IGX JSON Message Protocol",
    "type": "object",
    "properties": {
      "event": {
        "type": "string",
        "const": "set"
      },
      "data": {
        "type": "object",
        "patternProperties": {
          "^/.*$": { "type": ["string", "number", "boolean", "array", "object"] }
        },
        "additionalProperties": false
      }
    },
    "required": ["event", "data"],
    "additionalProperties": false
  }
}

```

Update Event

Sent by the device in response to a `get` event, carrying the subscribed field data. The `data` field is a dictionary containing the values for each path.

Example Message

```

{
  "event": "update",
  "data": {
    "/t1/probe/field/value": [
      [123, 1617981812.5],
      [124, 1617981812.6],
      ...
    ]
  }
}

```

JSON Schema

```

{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Update Event",
  "description": "Schema for update event in IGX JSON Message Protocol",
  "type": "object",
  "properties": {
    "event": {
      "type": "string",
      "const": "update"
    },
    "data": {
      "type": "object",
      "patternProperties": {
        "^/.*$": {
          "type": "array",
          "items": {
            "type": "array",
            "items": [
              { "type": ["string", "number", "boolean", "array", "object"] },

```

```

        { "type": "number" }
      ],
      "additionalItems": false
    }
  },
  "additionalProperties": false
},
"required": ["event", "data"],
"additionalProperties": false
}

```

Update Table Event

Sent by the device in response to a `get` event, carrying data for every subscribed table that has something new to report since the client's last `get`. Unlike the field `update` event, the `data` field is a dictionary keyed by table path, where each value is an object with an optional `config` key, an optional `rows` key, or both. A table is only included at all if it has a config change, new row data, or both.

Table Config

The `config` object carries the table's layout and metadata. Only fields that changed since the subscription's last update are included, except on the very first update after subscribing, when the full config is always sent. Commonly seen fields include `table_name`, `row_count`, `row_capacity`, `column_count`, `visible_column_count`, `column_names`, `column_types`, `column_units`, `column_hiddens`, `column_editables`, `highlight_enabled`, `highlight_row_begin`, `highlight_row_end`, and `focus_row`. `column_types` values are numeric type codes: `0` = Boolean, `1` = signed integer, `2` = unsigned integer, `3` = floating point.

ⓘ 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 \(see page 70\)](#) for a full worked example.

Table Rows

The `rows` value is a flat array of `[row_index, [cell0, cell1, ...]]` pairs. Cells within each row are ordered according to the table's column layout (respecting `include_hidden_columns` from the Config Event above), and a cell value of `null` means no data has been written to that cell yet. Because a table can generate a lot of row data at once (for example immediately after subscribing to a table that already has thousands of stored rows), a single `update_table` event may not include every pending row. Keep sending `get` events until no more new rows arrive.

Example Message

```

{
  "event": "update_table",
  "data": {
    "/fx4/data_acquisition": {
      "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, 0.131]]
    ]
}
}
}

```

JSON Schema

```

{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Update Table Event",
  "description": "Schema for update_table event in IGX JSON Message Protocol",
  "type": "object",
  "properties": {
    "event": {
      "type": "string",
      "const": "update_table"
    },
    "data": {
      "type": "object",
      "patternProperties": {
        "^/.+$": {
          "type": "object",
          "properties": {
            "config": { "type": "object" },
            "rows": {
              "type": "array",
              "items": {
                "type": "array",
                "items": [
                  { "type": "integer" },
                  { "type": "array" }
                ],
                "additionalItems": false
              }
            }
          }
        },
        "additionalProperties": false
      },
      "additionalProperties": false
    },
    "additionalProperties": false
  },
  "required": ["event", "data"],
  "additionalProperties": false
}

```

Update ID Event

This message will be sent by the server when it is assigning new shorter ID strings to full path names. These short names are used to save network bandwidth by not repeating potentially very long path names. This applies uniformly to field and table subscriptions alike: a subscribed table's path is remapped to a short ID exactly the same way a field's path is.

The data object of the event holds a series of key value pairs for all subscribed paths, where the key is the short name, and the value is the long path name. Clients should remember this mapping so that subsequent update messages can be remapped or translated back into the full path name and correctly handled. If, for whatever reason, a client forgets what IDs to use or needs to be reminded, they can send a `get_id` message to force another `update_id` message from the server.

Example Message

```
{
  "event": "update_id",
  "data": {
    "123": "/example/path/value",
    "abc": "/example/other/path/value",
    ...
  }
}
```

JSON Schema

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Update ID Event",
  "description": "Schema for update ID event in IGX JSON Message Protocol",
  "type": "object",
  "properties": {
    "event": {
      "type": "string",
      "const": "update_id"
    },
    "data": {
      "type": "object",
      "patternProperties": {
        "^[0-9a-zA-Z]+$": {
          "type": "string",
          "pattern": "^/([^/ ]+/*)*[^/ ]+$"
        }
      },
      "additionalProperties": false
    }
  },
  "required": ["event", "data"],
  "additionalProperties": false
}
```

Get ID Event

If a client would like to request the server to send an `update_id` message, they can use the `get_id` message to force an update. Once the server receives a `get_id` message, it will send an `update_id` message with all the subscribed paths and their corresponding IDs.

Example Message

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

JSON Schema

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
```

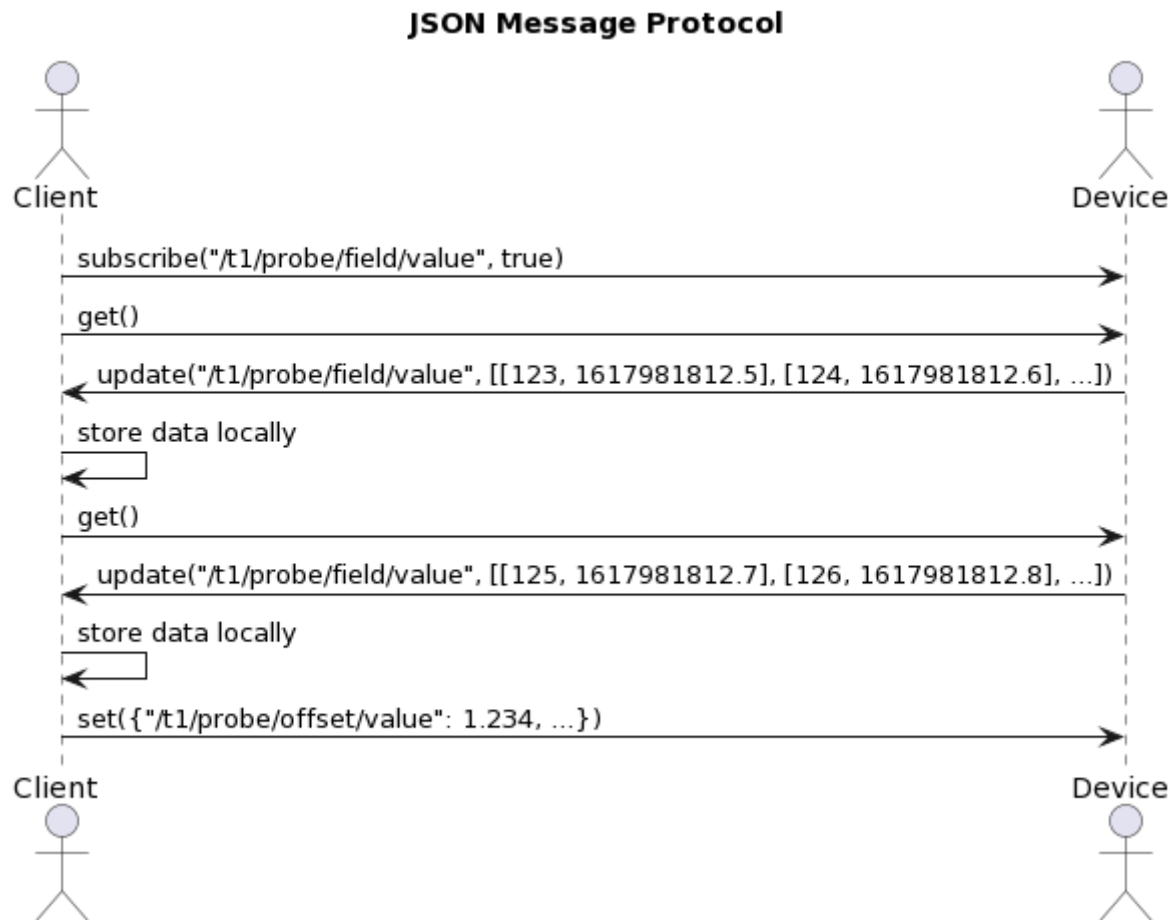
```
"title": "Get ID Event",
"description": "Schema for get ID event in IGX JSON Message Protocol",
"type": "object",
"properties": {
  "event": {
    "type": "string",
    "const": "get"
  }
},
"required": ["event"],
"additionalProperties": false
}
```

Typical Protocol Sequence

The basic protocol operates in the following way:

1. The client can optionally send a `config` event to the server to set any, non-default session configurations the client requires.
2. The client sends a `subscribe` and/or `subscribe_table` event to the server, specifying the data paths and/or tables it wants to subscribe to.
3. The client sends a `get` event to request new data from the device.
4. The device responds with an `update` event and/or an `update_table` event containing the requested data.
5. The client processes the received event(s) and stores the received data locally.
6. The client can repeatedly send `get` events to request more data from the device.

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



2 : JSON Event Timing Diagram

This JSON-based protocol provides a simple and flexible way to communicate between the client and the device over a WebSocket connection, allowing the client to subscribe to specific data paths ahead of time and receive updates as needed, conserving network resources. For a complete, worked example of using the table events specifically to capture data acquisition buffers, see the [IGX Data Acquisition Table Guide \(see page 70\)](#).

3.3.3 Suitable WebSocket Libraries

The following is a list of WebSocket client libraries that you can use to create connections and send messages. This guide will mostly use the Python library, but the other libraries will work in similar ways.

Language	Library Name	Description
JavaScript	WebSocket API ²⁶	Built-in WebSocket API in web browsers for real-time communication.
Python	websocket-client ²⁷	A WebSocket client library for Python with a focus on simplicity and ease of use.
Java	Java-WebSocket ²⁸	A full-featured WebSocket client library for Java applications.

26. <https://developer.mozilla.org/en-US/docs/Web/API/WebSocket>
27. <https://github.com/websocket-client/websocket-client>
28. <https://github.com/TooTallNate/Java-WebSocket>

Language	Library Name	Description
C	libwebsockets ²⁹	A lightweight, event driven WebSocket client and server library for C.
C++	WebSocket++ ³⁰	A lightweight, high-performance, and header only WebSocket client and server library for C++ applications.
C#	WebSocket4Net ³¹	A .NET WebSocket client implementation with support for various WebSocket protocol versions.
Go	gorilla/websocket ³²	A WebSocket client and server library for Go with a simple, idiomatic API.
Ruby	websocket-client-simple ³³	A simple and easy-to-use WebSocket client library for Ruby.
PHP	Ratchet ³⁴	A PHP library for building WebSocket servers and clients, enabling real-time communication in PHP applications.
Swift	Starscream ³⁵	A WebSocket client library for iOS, macOS, and tvOS, written in Swift.
LabVIEW	LabVIEW WebSockets API ³⁶	An unofficial WebSocket API for LabVIEW, providing WebSocket client and server functionality.

3.3.4 Python T1 Example

This guide will walk you through the process of setting up a WebSocket connection to collect, process, and store data in a CSV file using Python. This example uses the T1 device to collect magnetic field data.

Prerequisites:

- Python 3.x
- WebSocket client library: Install by running `pip install websocket-client`

Import the necessary Python libraries: `websocket`, `time`, `json`, and `csv`.

```
import websocket
import time
import json
import csv
```

Set up the device IP address, data collection time, and output file name.

```
ip = "192.168.55.239" # Device IP address
collection_time = 2.0 # Seconds to collect data
output_file = "t1_data.csv" # Data output file
```

Create a Python dictionary to store the data collected from the device. This will be used to temporarily store all the data points as they stream to the client.

```
# Database for storing collected data
```

29. <https://github.com/warmcat/libwebsockets>

30. <https://www.zaphoyd.com/websocketpp/>

31. <https://github.com/kerryjiang/WebSocket4Net>

32. <https://github.com/gorilla/websocket>

33. <https://github.com/ruby-jp/websocket-client-simple>

34. <http://socketo.me/>

35. <https://github.com/daltoniam/Starscream>

36. <https://www.ni.com/en-us/support/downloads/tools-network/download.websockets-api.html#374385>

```
database = {
    "/t1/probe/field/value": []
}
```

Create a WebSocket connection to the device using the `websocket.create_connection()` function. Passing `subprotocols` matches how IGX's own web client connects, and avoids clients that reject an unrequested `Sec-WebSocket-Protocol` response.

```
# Create the WebSocket, uses port 80 by default
ws = websocket.create_connection("ws://" + ip, subprotocols=["json.v1"])
```

Define functions to send and handle events, such as subscribing to data and requesting new data from the device. These helpers are simplified for the purpose of this document. In production code, it is a good idea to generalize functions like this further.

```
# Sends the device an event structure
# Optionally contains a payload called data
def sendEventData(event, data=None):
    # Convert dictionary to JSON and send
    ws.send(json.dumps({"event": event, "data": data}))

# Subscribe to the IO fields we are interested in
# In this case it is just the field value but there could be more
# The boolean value indicates whether the data should be buffered or not
# Buffered data means that all samples are sent to the client on a get event
# Unbuffered data means that only the most recent sample is sent on a get event
def sendSubscribeEvent():
    sendEventData("subscribe", {
        "/t1/probe/field/value": True
    })

# Request the device sends us the new data it has collected
# since the last time we sent a get event.
def sendGetEvent():
    # No data needed for the get event if you have already
    # previously sent the subscribe event message
    sendEventData("get")

# Response event handler, called every time we get a response
# from the device. Handles the processing of newly collected data
def onMessageEvent(event, data):
    # Check to make sure the response is an update event
    # Update events carry our subscription data
    if (event == "update"):
        # The dictionary contains all the values for each path
        for (path, values) in data.items():
            # Append the new values to the local database
            database[path] += values
        # Send another get event to request more data
        sendGetEvent()
```

Send a subscription event and a get event to start collecting data from the device. Once the first get event is sent, the following events will trigger from the `onMessageEvent()` function. It is important to not send a new get event until you have received an update from the server, as to not overload the device with requests.

```
# Send an initial subscription event and get event
# in order to start the collection process
sendSubscribeEvent()
```

```
sendGetEvent()
```

Collect data from the device for the specified duration using a while loop and the `onMessageEvent()` function. In this example we use time as a stopping condition, however it is perfectly acceptable to keep requesting new data forever.

```
# Remember the start time
start = time.time()

# Collect data for a given time
while time.time() - start < collection_time:
    # Wait for a responses from the device
    response = json.loads(ws.recv())
    # Process the received event and data
    onMessageEvent(response["event"], response["data"])</code>
```

Once data collection is complete, process the data and store it in a CSV file.

```
# Once we've finished collecting data we can process
# it however we like. In this case we write it to a CSV file
with open(output_file, "w", newline="") as file:
    writer = csv.writer(file, delimiter=",", quotechar="\"",
                        quoting=csv.QUOTE_MINIMAL)
    writer.writerow(["Values", "Timestamps"])

    value_pairs = database["/t1/probe/field/value"]

    for (value, time) in value_pairs:
        writer.writerow([value, time])

    print("Collected", len(value_pairs), "samples, written to", output_file)
```

Close the WebSocket connection using the `ws.close()` function.

```
# Close our connection
ws.close()
```

3.3.5 Conclusion

In conclusion, this user manual is designed to help you, as a programmer, understand how to use the WebSocket protocol and JSON messaging system to establish real-time communication between your Python script and an IGX device. By utilizing the WebSocket protocol, you'll be able to create efficient two-way communication for monitoring and controlling various parameters in industrial and IoT applications. The provided Python example serves as a guide, showing you how to set up a WebSocket connection, subscribe to data paths, handle incoming messages, and save the gathered data into a CSV file. With a clear understanding of these concepts and techniques, you can confidently adapt the example to fit your unique requirements and make the most of IGX devices in your projects. For working with Table data specifically, including data acquisition buffers, see the [Subscribe Table Event and Update Table Event sections above](#), and the [IGX Data Acquisition Table Guide \(see page 70\)](#).

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.

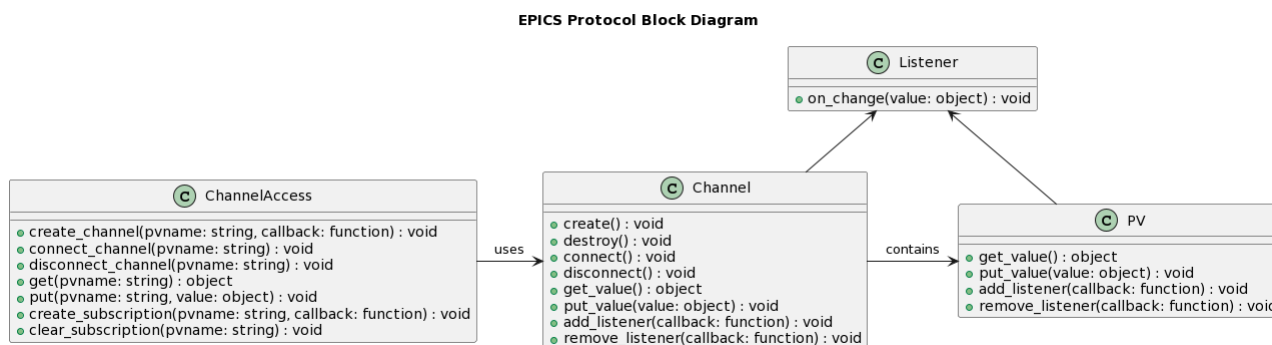
3.5 IGX EPICS Protocol Guide

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

EPICS primarily uses TCP for sending and receiving PV updates as well as for the Channel Access (CA) protocol. The CA protocol is used for managing communications between clients and servers, as well as for sending/receiving PV data.

EPICS also uses UDP for network broadcast, which is used for discovering other devices on the network that are running the CA protocol. UDP is also used for CA search, which allows clients to locate and connect to available servers on the network.

Protocol	Port	Description
TCP	5064	Used for sending/receiving PV updates.
TCP	5065	Used for Channel Access (CA) protocol.
UDP	5064	Used for network broadcast.
UDP	5065	Used for network broadcast and CA search.

It is important to note that some of these ports may need to be opened on your firewall in order for EPICS to function properly in a networked environment.

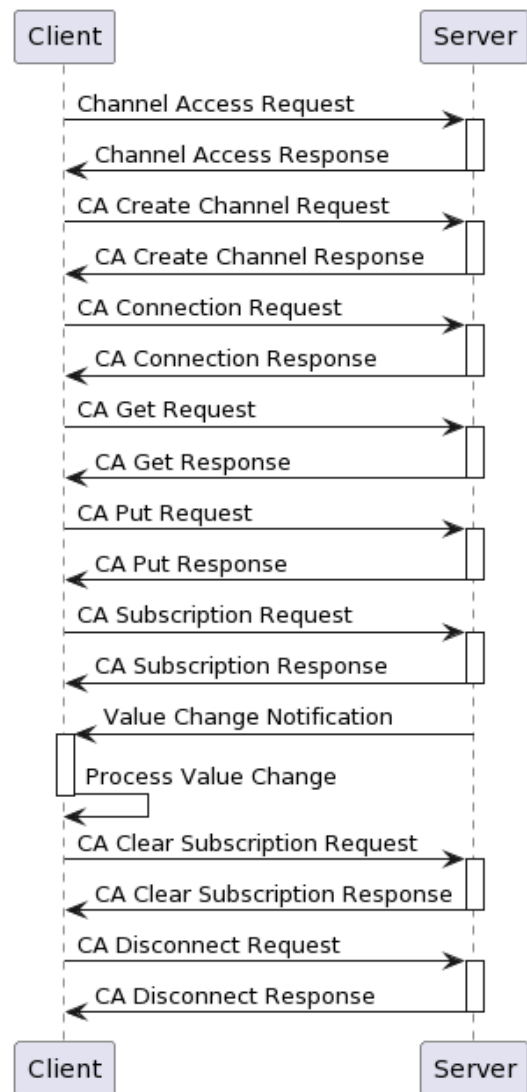
The timing diagram shows the sequence of events that occurs during a typical exchange between a client and server using the EPICS network protocol.

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.

EPICS Network Protocol Timing Diagram



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
<code>EPICS_CA_AUTO_ADDR_LIST=NO</code>	Disables automatic interface scanning (forces manual entry).
<code>EPICS_CA_SERVER_PORT=5064</code>	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. IGX devices operate as EPICS IOCs and expose parameters as PVs to allow integration without custom drivers.

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). It exposes device data and control points directly to EPICS clients without requiring custom EPICS drivers or support modules.

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.

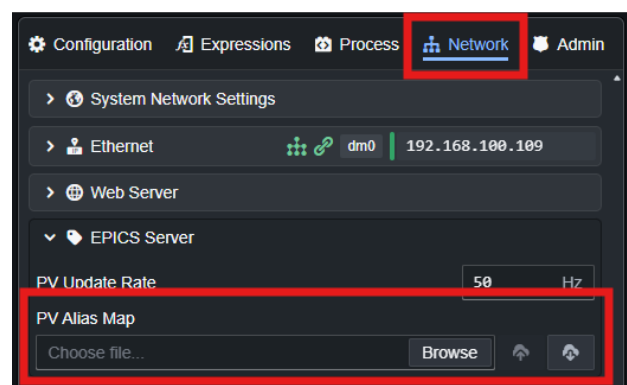
Alias XML Installation and Updating

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.



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, to access the heartbeat value field JSON file, clients can refer to the following path: `/io/heartbeat/value.json`. By following this path convention, clients can easily navigate the file structure and access the desired IO data.

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:
            data = json.load(file)
            print(f"Heartbeat value on {remote_node_name}: {data['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 a function `read_remote_heartbeat` that takes a remote node name as an argument, constructs the file path to the `/io/heartbeat/value.json` file on the remote node, and reads its contents. The script then prints the heartbeat value from the JSON data.

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
<code>ButtonIO</code>	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.
<code>ConfirmButtonIO</code>	Identical to <code>ButtonIO</code> 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.
<code>HoldButtonIO</code>	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 11) 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 70)) 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 \(see page 9\)](#) 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
<code>../state</code>	Corresponds to the current state of the module.	Read-only String
<code>../permit/user_command</code>	Turns on or off the power supply if the permit is granted by the interlock states.	Digital
<code>../command_voltage</code>	Sets the command voltage for the power supply output. Units are in volts.	Analog
<code>../monitor_voltage_internal</code>	The output voltage as measured by the internal sense circuit directly on the output. Units are in volts.	Read-only Analog
<code>../monitor_voltage_external</code>	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

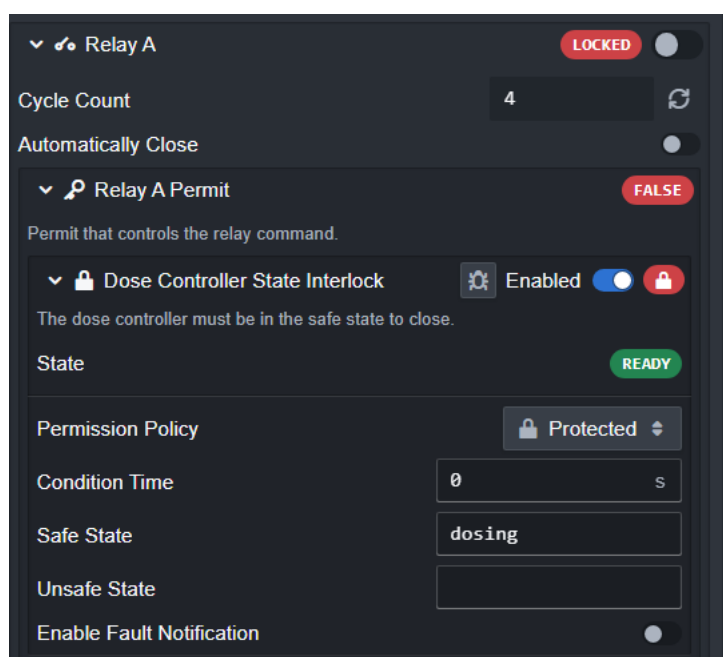
IO Path	Description
<code>.../permit/user_command</code>	Commands the relay open or closed. A true command will try to close the relay if the interlocks are granted, and false command will always open the relay.
<code>.../state</code>	The current state of the relay. Locked relays are open but cannot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

4.3.2 Advanced Control IO

IO Path	Description
<code>.../automatically_close</code>	When set to true, the relay will automatically close when the interlocks are granted. False by default.
<code>.../cycle_count</code>	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:

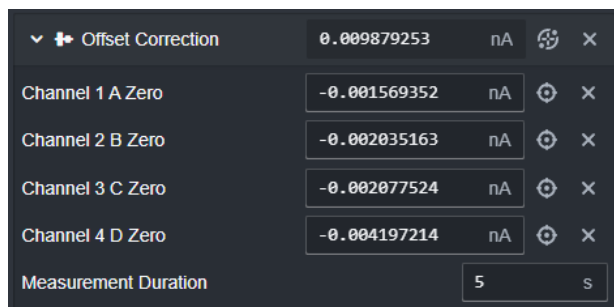
- Relay State - the relay state, “CLOSED” if the relay is closed, “OPENED” if the relay is opened, or “LOCKED” if the relay is under control by a process, such as Dose Controller.
- Relay Close - switch to enable or disable the relay if not LOCKED.
- Cycle Count - readback that counts the number of times 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.



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
<code>../value</code>	The current offset value. Changing this value will change the offset.	Analog

Name	Detail	Type
<code>../sequence/start_button</code>	Starts the automatic sequence. Once complete the offset will be set based on the current inputs.	Button Digital
<code>../sequence/stop_button</code>	Stop a running sequence if it is started. Does nothing if the sequence has not started.	Button Digital
<code>../clear_button</code>	Sets the current offset value to zero.	Button Digital
<code>../collecting</code>	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 \(see page 9\)](#) 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
<code>../state</code>	 Corresponds to the current state of the controller.
<code>../start_button</code>	 Start the controller if allowed by interlocks. Does nothing if interlocks are not granted.
<code>../pause_button</code>	 Pause the controller unconditionally. Does nothing if the controller is not dosing.
<code>../stop_button</code>	 Stops the controller unconditionally and permanently, preventing resuming the current session without an explicit reset command.
<code>../reset_button</code>	 Reset current dose delivery session if allowed. Does nothing unless the controller is stopped or completed.
<code>../dose_prescription</code>	 The prescription in dose units (MU or Gy) to convert into charge prescription. When this IO is modified, the <code>charge_prescription</code> is automatically updated based on the <code>dose_to_charge_expression</code> conversion expression.

IO Path	Description
<code>.../charge_prescription</code>	 The prescription based on charge units (nC). This IO is only writable when the <code>dose_prescription</code> is set to 0.
<code>.../charge_accumulation</code>	 The accumulated charge during the session.
<code>.../time_elapsed</code>	 The total elapsed time in seconds over the session since the first start command.
<code>.../time_active</code>	 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 plain-English string.

State	Description
<code>ready</code>	The controller is ready for accepting new prescription settings and can be started.
<code>dosing</code>	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.
<code>locked</code>	The controller is stopped and unable to start due to interlocks being in fault.
<code>paused</code>	The controller was paused after starting and is now waiting to be started again or stopped permanently.
<code>stopped</code>	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.
<code>completed</code>	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
<code>start_button</code>	Starts the controller if all the interlocks are ok, and the controller is in the <code>ready</code> or <code>paused</code> state.
<code>pause_button</code>	Pauses the controller if it is currently in the <code>dosing</code> state. The session can be resumed with another <code>start_button</code> command.
<code>stop_button</code>	Permanently stops the controller if it is in the <code>dosing</code> or <code>paused</code> 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.
<code>reset_button</code>	If the controller is in the <code>ready</code> , <code>stopped</code> , or <code>completed</code> 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	— Expression used to convert from dose (MU) to charge (nC).
.../threshold_current	— 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	— The amount of dose in MU per Q-Pulse. When this IO is modified, the <code>q_pulse_charge</code> is automatically updated based on the <code>dose_to_charge_expression</code> conversion expression.
.../q_pulse_charge	— The amount of charge in nC per Q-Pulse. This IO is only writable when the <code>q_pulse_dose</code> is set to 0.
.../q_pulse_min_width	— 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	— 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	— The measured position in mm along the X axis.
.../position_y	— The measured position in mm along the Y axis.
.../distance_from_nominal	— The distance in mm from the nominal position.
.../beam_sigma	— The expected beam size sigma in mm. Used to convert into mm units assuming a gaussian shaped beam.
.../nominal_x	— The nominal position in mm along the X axis.
.../nominal_y	— 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
<code>.../safety_test_pass_interlock</code>	<code>STATE_INTERLOCK</code> 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
<code>.../elapsed_time_limit_interlock</code>	<code>ELAPSED_TIME_LIMIT_INTERLOCK</code> The total elapsed time must be less than the given limit.
<code>.../depositing_current_range_interlock</code>	<code>DEPOSITING_CURRENT_RANGE_INTERLOCK</code> The dose rate, while the dose is enabled and detected, must be within a safe range.
<code>.../enabled_current_range_interlock</code>	<code>ENABLED_CURRENT_RANGE_INTERLOCK</code> The dose rate, while the dose is enabled, must be within a safe range.
<code>.../disabled_current_limit_interlock</code>	<code>DISABLED_CURRENT_LIMIT_INTERLOCK</code> The dose rate, while the dose is disabled, must be below the safe limit.
<code>.../safety_depositing_agreement_interlock</code>	<code>SAFETY_DEPOSITING_AGREEMENT_INTERLOCK</code> The safety input and depositing status must agree.
<code>.../safety_state_interlock</code>	<code>SAFETY_STATE_INTERLOCK</code> The safety input must be in the safe state.
<code>.../distance_from_nominal_interlock</code>	<code>DISTANCE_FROM_NOMINAL_INTERLOCK</code> While the dose is enabled, the beam position must be within the distance limit.
<code>.../hv_state_interlock</code>	<code>HV_STATE_INTERLOCK</code> 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
<code>.../safety_state_interlock</code>	<code>SAFETY_STATE_INTERLOCK</code> The safety input must be in the safe state.
<code>.../distance_from_nominal_interlock</code>	<code>DISTANCE_FROM_NOMINAL_INTERLOCK</code> While the dose is enabled, the beam position must be within the distance limit.
<code>.../hv_state_interlock</code>	<code>HV_STATE_INTERLOCK</code> 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
<code>/admin/mode</code>	 Sets the operating mode of the device. See the user manual for more information regarding the implications of this mode. Options: "develop", "product", "medical".
<code>/admin/device_type</code>	 The name of the type of device. For example, "T1", "FX4", "IX256", "MX1", and many others.
<code>/admin/serial</code>	 The Pyramid serial number for the device as a string. Typically, a number from 0-999999 but may include alphabetical characters in special circumstances.
<code>/admin/version</code>	 The IGX version name. "YY.MM.DDHHmm" format. This format is more human friendly but subject to change.
<code>/admin/version_int</code>	 The IGX version as an integer, useful for using in hashing or version control systems, not subject to change.
<code>/admin/clock/system_time_int</code>	 The system time in nanoseconds since January 1st, 1970, requires at least 64-bit representation to work.
<code>/admin/clock/system_time_string</code>	 The system time as a string. Displayed in a localized format that honors the configured time zone.

[illegible]

4.6.2 Permission Policy IO Interface

Typically, programmers will only want to access permission policy IO if they are implementing an 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
<code>../state</code>	  The current state of the permission policy. Options: “mutable”, “immutable”, “mutable_valid”, and “immutable_valid”.
<code>../hash_stale</code>	  True if the current hash has gone “stale” and has not yet been re-calculated to reflect the latest IO values.
<code>../hash_validated</code>	  True if the current hash is equal to the validated hash.
<code>../validate_button</code>	 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.
<code>../current_hash</code>	  The current SHA-256 hash of all the policy IO.
<code>../validated_hash</code>	  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
<code>.../enabled</code>	 Enable or disable the interlock.
<code>.../state</code>	  The state of the interlock if enabled and non latching
<code>.../latching</code>	 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 <code>unlatch()</code> function can be used to automate this
<code>.../condition_time</code>	 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
<code>.../fault_polarity</code>	 The Boolean value that causes a fault, default = False

RangelInterlockIO

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 captures a bounded buffer of samples from an ADC or other periodic source: a waveform, an event window, or a series of readings around a trigger. This guide covers its capture state machine, its full IO interface, how to trigger a single buffer capture, and how to approximate continuous acquisition by repeating captures back-to-back.

 This guide assumes familiarity with the [IGX WebSocket Protocol Guide \(see page 21\)](#), 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 to `collect_button` arms the table: if a start trigger is configured, `armed` goes high and the table waits for the trigger condition; if not, capture begins immediately. Once capture starts, `active` goes high and `sample_count` counts up on every row written. Capture stops automatically once `sample_count` reaches `buffer_size`, or earlier if a stop trigger fires, or immediately at any point if `cancel_button` is written. The full IO interface behind this state machine is listed below.

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
<code>armed</code>	True once the table is ready to collect data and is waiting for its start trigger (or, if no start trigger is configured, capture begins immediately and this is only true momentarily).	Read-only Digital
<code>active</code>	True while samples are actively being written to the table.	Read-only Digital
<code>sample_count</code>	The number of samples collected in the current buffer.	Read-only uint32
<code>buffer_period</code>	The amount of time the configured buffer spans at the current sample rate.	Read-only Analog (s)
<code>acquisition_progress</code>	Fraction of the current capture completed, from 0.0 to 1.0. Also exposes the standard progress sub-fields (<code>percent_float</code> , <code>time_remaining</code> , and so on).	Read-only Progress
<code>acquisition_gated</code>	True when gating is currently suppressing sample writes (see <code>gating_enabled</code> below).	Read-only Digital
<code>external_elapsed_time</code>	Not written over the API. If a component wires an external clock source into this channel (non-zero), it replaces the table's own internal elapsed-time computation for 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

These live under the table's `config` sub-node, which is hidden from the web GUI's navigation tree but is addressable the same as any other IO, for example `.../data_acquisition/config/row_capacity`.

Name	Detail	Type
<code>config/row_capacity</code>	Maximum number of rows the table can currently hold, derived from <code>max_data_size_bytes</code> and the current column count.	Read-only uint32
<code>config/row_count</code>	Current number of rows stored in the table.	Read-only uint32
<code>config/column_count</code>	Total number of active columns, including the time column.	Read-only uint32
<code>max_data_size_bytes</code>	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.	Analog (Bytes)
<code>memory_allocated</code>	True once memory has actually been allocated for the table.	Read-only Digital
<code>allocated_memory_size</code>	Size of memory actually allocated for the table.	Read-only Analog (Bytes)

5.2.3 Configurable Parameters

Name	Detail	Type
<code>buffer_size</code>	Target number of rows to collect; capture stops automatically once <code>sample_count</code> reaches this value. Clipped to <code>config/row_capacity</code> when the table arms.	Analog uint32
<code>gating_enabled</code>	Enables gating of the acquisition.	Digital
<code>gating_expression</code>	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)
<code>reset_counters_on_clear</code>	When enabled (the default), any selected counter-type channel's running total is reset whenever the table is cleared or a new acquisition is armed.	Digital
<code>acquisition_channels</code>	The set of channels included as data columns. VTree name is <code>standalone_channels</code> for backward config compatibility, but the label and current behavior are "Acquisition Channels."	Symbol multi-select

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
<code>enabled</code>	Enables this trigger. A disabled start trigger means collection begins immediately upon arming; a disabled stop trigger means collection only ends when <code>buffer_size</code> is reached or <code>cancel_button</code> is written.	Digital
<code>source</code>	The IO path this trigger watches. Any IO on the device can be used, not just a dedicated trigger input.	Symbol select
<code>mode</code>	Rising Edge fires the moment <code>source</code> 's value crosses from below <code>threshold</code> up to at or above it. Falling Edge fires the moment it crosses back down through <code>threshold</code> .	Select (rising/falling)
<code>threshold</code>	The value <code>source</code> 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 \(see page 56\)](#) 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
<code>collect_button</code>	Arms the table, starting the capture sequence.	Button
<code>cancel_button</code>	Immediately stops an armed or in-progress capture.	Button
<code>universal_button</code>	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)
<code>clear_button</code>	Inherited from the base <code>Table</code> class, labeled "Clear Table." Wipes all rows currently stored in the table. Distinct from stopping a capture: cancelling does not by itself erase previously collected 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
<code>individual_channels_enabled</code>	Labeled "Acquire Individual Channels" in the UI. When enabled, every active channel on this ADC is added as a table column automatically, without needing to select them individually in <code>acquisition_channels</code> . Extra computed values the ADC provides (sums, curve fits, and similar) still need to be selected individually in <code>acquisition_channels</code> regardless of this setting.	Digital

5.3 Capturing a Buffer Once

The single-shot sequence is: clear any old data, configure the buffer, arm, wait for completion. The examples below use `/fx4/data_acquisition`, the Data Acquisition table on an FX4 device, but the same event shapes apply to any device's table at its own path. This is a straightforward extension of the pattern in the [WebSocket Protocol Guide \(see page 21\)](#)'s Python example: subscribe to the table for row data, subscribe to `active` separately as an ordinary field (capture state is not part of a table's config, see the note above), configure and arm the capture with `set` events, then watch for `active` to go back to false.

```
import websocket, json

path = "/fx4/data_acquisition"
ws = websocket.create_connection("ws://" + ip, subprotocols=["json.v1"])

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

# Subscribe to the table for row/column data, and to "active" as an
# ordinary field: capture state isn't part of a table's config.
send("subscribe_table", {path: True})
send("subscribe", {path + "/active/value": True})

# Clear any previous data, then configure and arm the capture
send("set", {
    path + "/clear_button/value": True,
    path + "/buffer_size/value": 1000,
})
send("set", {path + "/collect_button/value": True})

# Poll for updates until the capture completes (active goes back to false)
active = True
rows = []
while active:
    send("get")
    response = json.loads(ws.recv())
    event, data = response["event"], response["data"]

    if event == "update_table" and path in data:
        rows += data[path].get("rows", [])
    elif event == "update" and path + "/active/value" in data:
        # Buffered field updates arrive as a list of [value, timestamp] pairs
        for value, timestamp in data[path + "/active/value"]:
            active = value

print("Captured", len(rows), "rows")
```

```
ws.close()
```

A few notes on this pattern:

- The first `update_table` received after `subscribe_table` always includes the table's full config and every row already stored, so clearing the table first (as above) avoids processing stale data left over from a previous capture.
- `active`, `armed`, and `sample_count` arrive through ordinary `update` events on their own subscribed field paths, not inside the table's `update_table` config. Subscribe to whichever of them your client needs to watch.
- Row delivery is batched (roughly every 100 ms server-side) and throughput-limited per message, so a large buffer may take several `get / update_table` round-trips to fully drain even after the capture itself has completed.

5.4 Pseudo-Continuous Acquisition

A Data Acquisition table always stops itself once `sample_count` reaches `buffer_size`, or a stop trigger fires. There is no built-in "keep capturing forever" flag on it. To approximate continuous acquisition, re-arm the table as soon as one capture completes, producing a rapid sequence of back-to-back, non-overlapping buffers rather than one uninterrupted stream. This is what "pseudo-continuous" means here: repeated single-shot captures, not literal continuous streaming.

Watch for the separately-subscribed `active` field transitioning back to `false`, and immediately send another `collect_button` set to start the next buffer:

```
send("subscribe_table", {path: True})
send("subscribe", {path + "/active/value": True})

was_active = False
while collecting:
    send("get")
    response = json.loads(ws.recv())
    event, data = response["event"], response["data"]

    if event == "update_table" and path in data:
        rows = data[path].get("rows", [])
        if rows:
            handle_buffer_rows(rows) # e.g. append to a file, plot, etc.

    elif event == "update" and path + "/active/value" in data:
        for is_active, timestamp in data[path + "/active/value"]:
            if was_active and not is_active:
                # previous buffer finished, start the next one
                send("set", {path + "/clear_button/value": True})
                send("set", {path + "/collect_button/value": True})
            was_active = is_active
```

5.4.1 Practical Considerations

- **There will be gaps between buffers.** Clearing, re-arming, and (if configured) waiting for a start trigger all take a small but non-zero amount of time, so consecutive buffers are not sample-contiguous.
- **Size buffers with round-trip time in mind.** Very small `buffer_size` values re-arm extremely quickly, which can generate a large number of WebSocket messages per second; very large values reduce message overhead but increase the delay before you see a given buffer's data (due to the ~100 ms row-batching interval and per-message row limits mentioned above).

- **Clear between captures** if you don't want each buffer's `update_table` payload to include leftover rows from the previous capture beyond `buffer_size`.

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	<code>True</code> 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	<code>True</code> 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](#)³⁷. 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 \(see page 9\)](#) 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](#)³⁸
- [SFTP - Secure File Transfer Protocol](#)³⁹
- [HTTP - Hypertext Transfer Protocol](#)⁴⁰
- [SSH - Secure Shell](#)⁴¹
- [Qnet - QNX Networking Protocol](#)⁴²

37. https://www.qnx.com/developers/docs/7.1/#com.qnx.doc.neutrino.sys_arch/topic/qnet.html

38. <https://www.json.org/>

39. https://en.wikipedia.org/wiki/SSH_File_Transfer_Protocol

40. https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol

41. https://en.wikipedia.org/wiki/Secure_Shell

42. https://www.qnx.com/developers/docs/6.5.0_sp1/index.jsp?topic=%2Fcom.qnx.doc.neutrino_prog%2Fqnet.html