

IGX

IGX - Programmer Manual

Table of Contents

1	Introduction	5
1.1	Approvals	5
1.2	Revisions	5
2	IGX Network Protocols	6
2.1	Overview of IGX Protocols.....	6
2.1.1	HTTP	6
2.1.2	WebSocket	6
2.1.3	EPICS	7
2.1.4	SFTP	7
2.1.5	Qnet	8
2.2	IGX HTTP Protocol Guide	9
2.2.1	IGX HTTP Server	9
	HTTP Quick Start.....	9
	Suitable HTTP Libraries	10
2.2.2	HTTP Basics	11
	TCP Connection.....	11
	HTTP URLs	12
	HTTP Request Structure	13
	HTTP Response Structure	13
2.2.3	HTTP Python Examples	14
	Sending a GET Request	14
	Sending a PUT Request	15
2.2.4	Postman for HTTP(S) Testing.....	17
	Sending a GET Request	17
	Sending a PUT Request	17
2.2.5	PLC HTTP Programming	18
	Protocol Gateways	19
	Custom Code Solutions	19
2.2.6	Summary.....	21
2.3	IGX WebSocket Protocol Guide	22
2.3.1	WebSocket Overview	22
	IGX JSON Message Protocol.....	24

Suitable WebSocket Libraries	29
2.3.2 Python T1 Example.....	31
2.3.3 Conclusion	33
2.4 IGX EPICS Protocol Guide	34
2.4.1 EPICS Overview	34
EPICS Network Protocols and Ports	35
Choosing EPICS for Your Control System	37
2.4.2 IGX EPICS Interface	38
Handling Multiple Devices	38
EPICS Utility Programs	39
2.4.3 Python Examples	41
Python Library Overview	41
Getting a PV (IO) Value	43
Putting a PV (IO) Value	43
Zeroing a T1 Probe.....	44
Polling Heartbeat	45
Subscribing to Heartbeat	46
2.4.4 Conclusion	48
2.5 IGX SFTP Protocol Guide	48
2.5.1 Use SFTP for IGX IO Data	48
Accessing IGX IO Data via SFTP	48
Python Example	49
2.6 IGX Qnet Protocol Guide	50
2.6.1 Qnet Overview.....	50
Network Protocol.....	50
Device Discovery.....	50
The Net Directory.....	51
Network Packet Passing.....	51
File Exposure between Nodes	52
2.6.2 Integrating Qnet with IGX	52
Accessing IO Data through Qnet	52
File Path Conventions in IGX	52
2.6.3 Python Qnet Example	53
3 IGX File Format Specifications	55

3.1 IGX JSON IO Files.....	55
3.1.1 Introduction	55
Example JSON	55
Field Types	55
Index JSON Files	56
Field JSON Files.....	57
JSON Schema	57
Availability of Protocols	58
Helpful Links.....	58
3.2 IGX XML Configuration Files	59
3.2.1 Overview	59
Structured Node Format	59
Possible Field Types (Attributes)	59
Possible Node Types	61
Possible IO Types	61
A Simple Example	62
4 IGX Standard IO Interfaces	63
4.1 IGX Dose Controller IO Interface	63
4.1.1 Overview	63
Interface IO.....	63
4.1.2 Use Case Examples.....	65
4.2 IGX High Voltage IO Interface	66
4.2.1 Overview	66
Interface IO.....	66
Module States	67
Safety Interlocks.....	67
Internal Voltage Sense Circuit	67
External Voltage Sense Circuit	67
4.2.2 Use Case Examples.....	68
Setting and Enabling High Voltage Output	68
Reading Internal Voltage Values.....	68
Monitoring External Voltage Values (Optional)	68

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.

1.1 Approvals

This document has been reviewed and approved as follows.



Document Control

No reviewers assigned.

1.2 Revisions

Version	Description	Saved by	Saved on	Status
---------	-------------	----------	----------	--------

2 IGX Network Protocols

2.1 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.

2.1.1 HTTP

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.

Weaknesses

- Lots of overhead per transaction.
- Lacks support for lossless buffered data.

See the full guide here: [IGX HTTP Protocol Guide](#)

2.1.2 WebSocket

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

- Streaming lossless buffered data.

- Can manage hundreds of IO through a single TCP connection.
- Strong community of developers and libraries.

Weaknesses

- Can be complex to implement without pre-made libraries.

See the full guide here: [IGX WebSocket Protocol Guide](#)

2.1.3 EPICS

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.

Strengths

- Standard in some big-experiment facilities.
- Simple to use with the given libraries.

Weaknesses

- Libraries are hard to compile for some platforms.
- Lack of community support.
- Lots of pitfalls in the implementation process.

See the full guide here: [IGX EPICS Protocol Guide](#)

2.1.4 SFTP

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.

Weaknesses

- High latency per transaction.
- Low overall network bandwidth capability.

See the full guide here: [IGX SFTP Protocol Guide](#)

2.1.5 Qnet

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

- Incredible performance and latency.
- Very easy to implement and use.

Weaknesses

- Requires the QNX operating system.
- Currently lacks support for buffered data.

See the full guide here: [IGX Qnet Protocol Guide](#)

2.2 IGX HTTP Protocol Guide

2.2.1 IGX HTTP Server

All IGX devices come with a high-performance [HTTP](#) 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](#) files for storing IO data makes it easy to integrate with various programming languages and data exchange formats.

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](#) to GET field value.

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

use [cURL](#) to PUT a field value.

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

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 GET field value.

```
import requests
print(requests.get("http://<IP ADDRESS>/io/heartbeat/value.json").json())
```

Suitable HTTP Libraries

In this guide we will be using [Python](#) and the [requests library](#) 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 likely many more out there.

Language	Library	Notes
Python	requests	Very simple to use library, used at Pyramid for internal testing tools. Uses urllib3 under the hood.
Python	urllib3	Lower-level library, useful if you want to have more control and don't want extra code.
JavaScript	fetch	Newer built-in JS function for making HTTP requests.
JavaScript	XMLHttpRequest	The old way of making HTTP requests, useful if your environment doesn't support modern JavaScript.
C	libcurl	A very complete library that includes HTTP and other file transfer protocols.
C++	cpr	A C++ wrapper for libcurl that is inspired by the Python library requests.
C++	HTTPRequest	Single header file implementation. Great for small projects or if you need a simple build process.

Language	Library	Notes
C#	HttpClient	Newer and preferred way to make HTTP requests in .NET
C#	HttpWebRequest	The older way of making an HTTP request.
Java	Java 11 HttpClient	Newer built-in library for HTTP.
Java	Apache HttpClient	Older but still a very popular library.
LabView	GET VI	Built-in LabView HTTP VI to send GET requests.
LabView	PUT VI	Built-in LabView HTTP VI to send PUT requests.

2.2.2 HTTP 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.

We covered 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.

Overall, HTTP is an essential tool for developers working with IGX, as it provides a simple and efficient way to interact with the control system and retrieve or modify data.

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 messages 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.

By understanding the basics of 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.

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](#) library for Python does this automatically, so you don't have to worry about it in these examples. 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.
- 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.
- 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.

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

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

2.2.3 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.

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).jso
n()

# 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.

2.2.4 Postman for HTTP(S) Testing

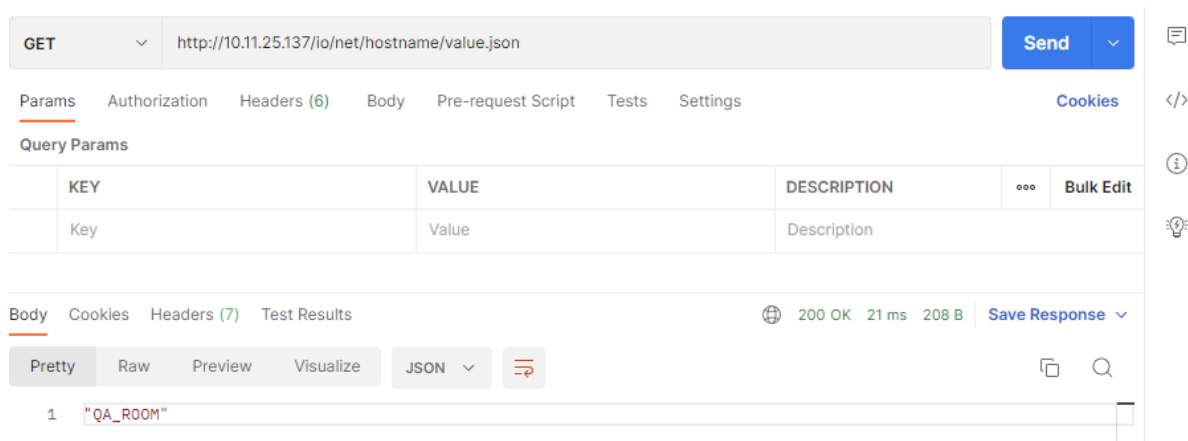
Postman is a powerful tool for sending and receiving HTTP messages with a user-friendly GUI interface. It can be invaluable for debugging your code and testing new queries without the hassle of programming. You can download Postman from the following link:

<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.

PUT ▼ http://10.11.25.137/io/net/hostname/value.json Send ▼

Params Authorization Headers (9) Body ● Pre-request Script Tests Settings Cookies

Headers 8 hidden

	KEY	VALUE	DESCRIPTIO	...	Bulk Edit	Presets
<input checked="" type="checkbox"/>	Content-Type	application/json				
	Key	Value	Description			

Body Cookies Headers (6) Test Results 200 OK 107 ms 192 B Save Response ▼

Pretty Raw Preview Visualize JSON ▼ 🔍

```

1  {
2    "status": "success"
3  }
```

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

PUT ▼ http://10.11.25.137/io/net/hostname/value.json Send ▼

Params Authorization Headers (9) Body ● Pre-request Script Tests Settings Cookies

none form-data x-www-form-urlencoded raw binary GraphQL JSON ▼ Beautify

```

1  {"QA_ROOM_1"}
```

Body Cookies Headers (6) Test Results 200 OK 107 ms 192 B Save Response ▼

Pretty Raw Preview Visualize JSON ▼ 🔍

```

1  {
2    "status": "success"
3  }
```

5. Click the Send button and observe the response.

2.2.5 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.

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](#)
- [Moxa NPort Gateway](#)
- [Red Lion Data Station](#)

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

Custom Code Solutions

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

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

Writing custom code can offer more flexibility and control over the communication process, but it also requires more development effort and maintenance. It is important to

carefully consider the requirements of the application and the capabilities of the PLC before deciding to write custom code.

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

```
#include <arpa/inet.h>
#include <netinet/in.h>
#include <sys/socket.h>
#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];
    sprintf(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.

2.2.6 Summary

In conclusion, HTTP (Hypertext Transfer Protocol) is the backbone of modern web applications and APIs, allowing clients to send requests to servers and receive responses in a standard format. IGX provides an HTTP interface that allows clients to read and write IO data on devices running the IGX control system.

HTTP requests and responses have two main parts, the header and the body, which are separated by new lines. The header contains information about the request or response, such as the URL, method, status code, and content type. The body contains the actual data being transferred, which can be in various formats depending on the API and the use case.

To interact with IGX over HTTP, developers can use various tools and libraries, including cURL, Python's requests library, Postman, and more.

Overall, HTTP is a fundamental technology for building web applications and APIs, and it's essential for any developer working in this space to have a good understanding of how it works and how to use it effectively.

2.3 IGX WebSocket Protocol Guide

2.3.1 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.

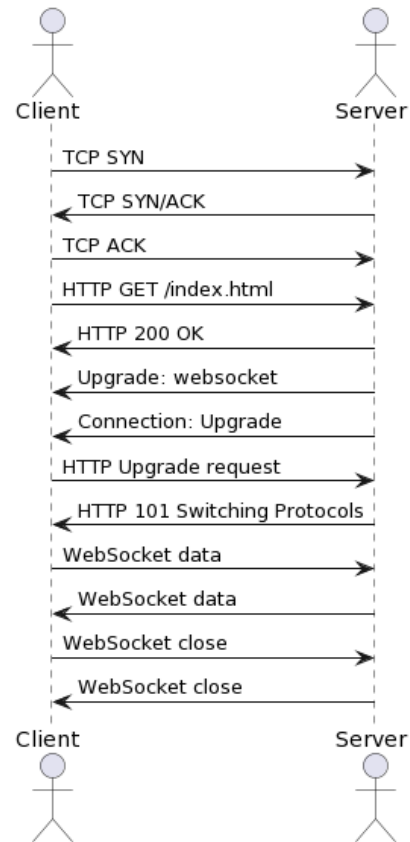
In the context of IGX, the WebSocket protocol provides a convenient and efficient way to exchange data between a client (such as a Python script) and an IGX device. The protocol is particularly well-suited for use cases where continuous, real-time updates are required, such as monitoring sensor values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

```
ws = websocket.create_connection("ws://" + ip)
```

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.

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.

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.

Example:

```
{
  "event": "subscribe",
  "data": {
```



```

    "/t1/probe/field/value": true
  }
}

```

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
}

```

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" event, no additional data field is required for this message. After receiving a get event, the IGX service will send an update event with the requested 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 an update message. This is to prevent the server from being overloaded with incoming messages.

Example:

```

{
  "event": "get"
}

```

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. It is wise to subscribe to fields that you plan on modifying, so that you can confirm the value has changed in later update messages.

Example:

```

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

```

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 data. The "data" field is a dictionary containing the values for each path.

Example:

```

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

```

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  
}
```

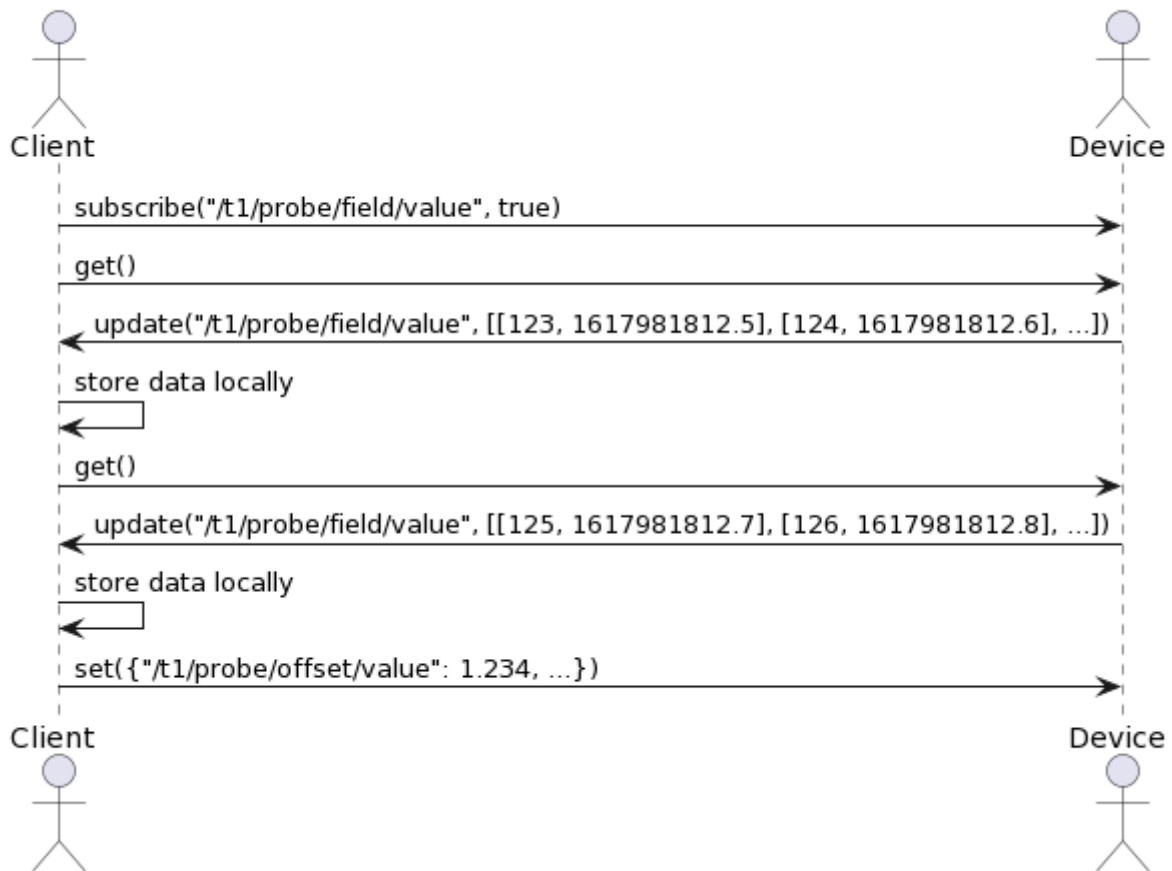
Protocol Sequence

The basic protocol operates in the following way:

1. The client sends a "subscribe" event to the device, specifying the data paths it wants to subscribe to and whether the data should be buffered.
2. The client sends a "get" event to request new data from the device.
3. The device responds with an "update" event containing the requested data.
4. The client processes the "update" event and stores the received data locally.
5. 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:

JSON Message Protocol



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.

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	Built-in WebSocket API in web browsers for real-time communication.

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

2.3.2 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
database = {
    "/t1/probe/field/value": []
}
```

Create a WebSocket connection to the device using the `websocket.create_connection()` function.

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

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"])

```

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()

```

2.3.3 Conclusion

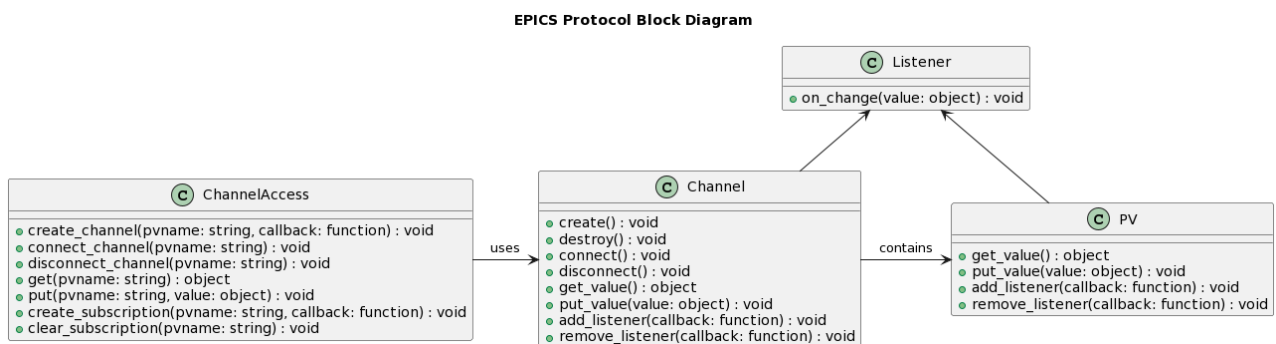
In conclusion, this user manual is designed to help you, as a programmer, understand how to use the WebSocket protocol and JSON messaging system to establish real-time communication between your Python script and an IGX device. By utilizing the WebSocket protocol, you'll be able to create efficient two-way 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.

2.4 IGX EPICS Protocol Guide

2.4.1 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.
UDP	5066	Used for network broadcast and CA search.
UDP	5067	Used for 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.

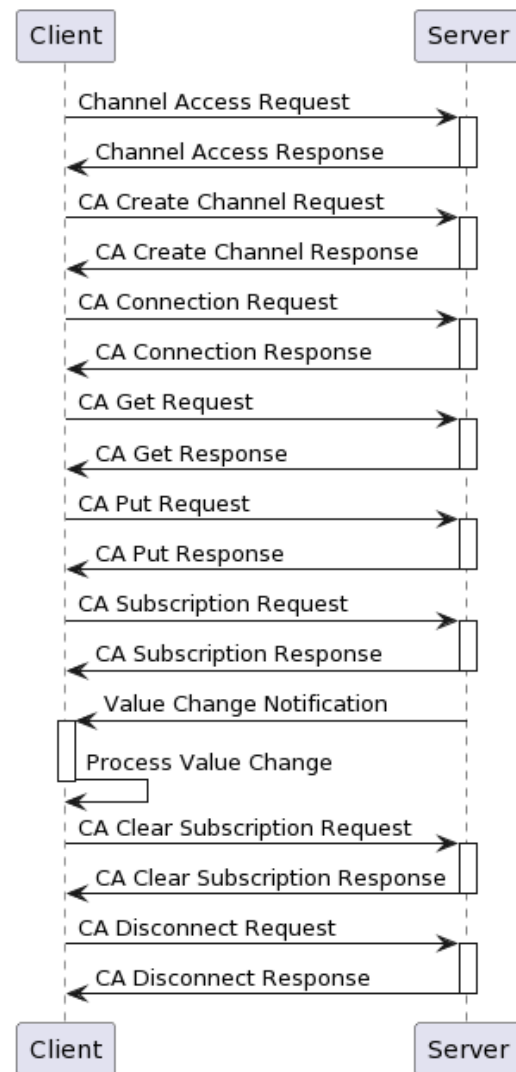
The timing diagram shows the sequence of events that occurs during a typical exchange between a client and server using the EPICS network protocol.

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 control systems leverage a powerful network-based discovery mechanism, which is designed to identify and communicate with devices on the local network without the need for manual IP address configuration. This auto-discovery process is facilitated by the Channel Access (CA) protocol, which utilizes User Datagram Protocol (UDP) broadcasts to locate and establish connections with available devices.

When a client attempts to connect to a Process Variable (PV) on an EPICS Input/Output Controller (IOC) server, the client sends a UDP broadcast request containing the PV name. All IOCs on the network receive the broadcast and compare the requested PV name with their internal PV lists. If an IOC has the specified PV, it responds to the client with its IP address and the connection is established using the Transmission Control

Protocol (TCP). This exchange occurs automatically, enabling efficient and seamless communication between the client and multiple devices within the control system.

The EPICS discovery mechanism not only simplifies device connectivity, but it also enhances the overall performance and reliability of the control system. By eliminating the need for manual IP address specification, EPICS reduces the potential for human error and ensures that devices can be easily added, removed, or replaced as needed. Furthermore, the broadcast nature of the discovery process ensures that clients can locate devices even if their IP addresses change, providing a robust and flexible control system environment.

Using EPICS Across Network Barriers

EPICS can be used across network barriers, such as firewalls or across different subnets, by configuring EPICS to use gateway or proxy servers. These servers act as intermediaries, allowing communication between EPICS clients and servers that are not directly connected.

One approach is to use the Channel Access Gateway (CAG) which enables secure communication between EPICS clients and servers that are on different subnets or behind firewalls. The CAG allows clients to connect to the gateway and the gateway will route the requests to the appropriate server. This is done through TCP/IP communication over port 5064 for the CAG server and port 5065 for the CAG client.

Another approach is to use the EPICS Channel Access over Secure Shell (CASSH) which uses SSH port forwarding to create a secure tunnel between the client and the server. This allows clients to access EPICS servers without needing to open ports on the firewall or set up a gateway. However, CASSH requires that an SSH server be installed on the gateway machine and the clients must have SSH client software installed.

Additionally, using the EPICS Gateway Application (GATE) allows communication between two EPICS networks that are isolated by a firewall or across different subnets. The GATE software operates as a proxy, allowing clients on one network to communicate with servers on the other network. This is done through TCP/IP communication over port 2064 for the GATE server and port 2065 for the GATE client.

Choosing EPICS for Your Control System

EPICS is a powerful and widely used control system framework that offers numerous benefits for managing and controlling devices in various scientific and industrial applications. If your control system already employs EPICS, it can be an excellent choice for managing and controlling your IGX devices.

However, if your control system does not currently utilize EPICS, you may want to investigate alternative communication methods before committing to EPICS. Some alternatives might be easier to set up and require less library support, making them more suitable for specific use cases or smaller-scale projects.

Ultimately, the choice of communication method should be based on your project requirements, existing infrastructure, and the level of expertise within your team. EPICS offers a convenient way to access field values on IGX devices without needing to develop custom drivers, but it is essential to carefully evaluate the pros and cons of using EPICS compared to other communication options available.

2.4.2 IGX EPICS Interface

An IGX device functions as an Input/Output Controller (IOC) server within the EPICS (Experimental Physics and Industrial Control System) framework. As an IOC server, the IGX device enables seamless integration with EPICS-based control systems without requiring custom drivers.

In the EPICS framework, an IOC is a device that hosts one or more Process Variables (PVs) and handles the communication between the control system and the actual hardware. PVs represent the properties of the controlled devices, such as sensor readings, control parameters, and system status. By functioning as an IOC server, an IGX device exposes its PVs to the control system, allowing the EPICS clients to monitor and control the device's parameters and functions.

The IGX device, when acting as an IOC server, handles the following tasks:

1. **Exposing PVs:** The IGX device automatically generates PV names corresponding to the field paths of the device parameters. These PVs can be accessed by EPICS clients to read or write data.
2. **Communication:** The IGX device communicates with the EPICS clients using the Channel Access (CA) protocol, a high-performance communication protocol designed for use within the EPICS framework. This allows for efficient data exchange between the device and the control system.
3. **Device management:** As an IOC server, the IGX device takes care of handling the device-specific operations, such as reading sensor data, controlling actuators, and managing internal settings.

By functioning as an IOC server, the IGX device simplifies integration with EPICS-based control systems, allowing users to focus on developing their applications without worrying about creating custom drivers for their devices. This approach also provides the benefits of the EPICS ecosystem, such as a robust and flexible architecture, a wide range of supported devices, and an active community of developers and users.

Handling Multiple Devices

In cases where you have multiple IGX devices of the same type within your control system, it is important to differentiate between them to ensure proper communication and control. To achieve this, you can use a unique identifier, such as the IP address, device

serial number, or hostname, and prepend it to the corresponding channel name. This allows you to target specific devices when reading or writing PV values.

For example, using IP addresses:

Device 1 - IP Address: `192.168.0.5` Channel name: `192.168.0.5:/device/sub_module/voltage/value`

Device 2 - IP Address: `192.168.0.6` Channel name: `192.168.0.6:/device/sub_module/voltage/value`

Alternatively, using hostnames or serial numbers:

Device 1 - Hostname: `my-device-1` Channel name: `my-device-1:/device/sub_module/voltage/value`

Device 2 - Hostname: `my-device-2` Channel name: `my-device-2:/device/sub_module/voltage/value`

By using this approach, you can effectively manage multiple IGX devices within the EPICS control system, ensuring accurate and reliable communication between the devices and your applications. Utilizing the device serial number or hostname as an identifier can be particularly helpful in cases where IP addresses might change due to network configurations or device reassignments. This allows for a more stable and consistent identification method for your IGX devices within the control system.

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
```

2.4.3 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...
```

2.4.4 Conclusion

EPICS is an open-source control system used for scientific instruments that allows efficient communication between hardware devices and software applications. Its core is the Channel Access protocol, which allows communication between servers (Input/Output Controllers, or IOCs) and clients. EPICS leverages a network-based discovery mechanism that identifies and communicates with devices on the local network without manual IP address configuration. The IGX device acts as an Input/Output Controller (IOC) server within the EPICS framework, enabling seamless integration without requiring custom drivers. By utilizing EPICS, users can monitor and control their devices' parameters and functions easily.

2.5 IGX SFTP Protocol Guide

2.5.1 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.

2.6 IGX Qnet Protocol Guide

2.6.1 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.

2.6.2 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.

2.6.3 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.

3 IGX File Format Specifications

3.1 IGX JSON IO Files

3.1.1 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.
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](#). 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](#) 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](#)
- [SFTP - Secure File Transfer Protocol](#)
- [HTTP - Hypertext Transfer Protocol](#)
- [SSH - Secure Shell](#)
- [Qnet - QNX Networking Protocol](#)

3.2 IGX XML Configuration Files

3.2.1 Overview

IGX uses an [XML](#) file to know how to configure itself on startup. The default name for this file is `system.xml`. IGX XML is naturally minimalist, meaning that anything that comes by “default” does not need to be defined in the XML. Only if your configuration breaks from the normal case, do you need to modify the `system.xml` file.

This XML file is stored by default at `/root/config/system.xml`.

Structured Node Format

The format of the XML file follows standard XML formatting rules. The top level node must be `root`.

```
<root>
  <node name="example" />
</root>
```

The parent child relationship in the XML mimicked in the runtime IGX structure. So if you define a new node underneath another node, IGX will make that node a child of the parent node.

```
<root>
  <node name="parent" >
    <node name="child" />
  </node>
</root>
```

Fields can be assigned through node attributes. Fields allow you to define properties for nodes and tell the system and GUI how it should handle this addition.

```
<root>
  <node name="parent" detail="My custom node that contains a custom XML IO" >
    <analog_io name="my_xml_io" label="My XML IO" units="pA" value="0.1" />
  </node>
</root>
```

Possible Field Types (Attributes)

The following is all the possible field types you can define on a node. The most commonly used fields are, name, label, value, readonly, units, and format.

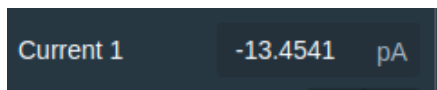


Field	Type	Required	Notes
name	string	Yes	Name of the node, must be unique among sibling nodes. Must not be the same as any field name. For example name can not equal “name” or “label”.
alias	string	No	A unique identifier for the IO, used by EPICS server for PV name.
label	string	No	Human friendly name for the node, used by GUIs.
detail	string	No	Brief description of the node, used by the GUI and automated report generation.
hidden	bool	No	True if this node is too complicated for the average user and should be hidden by the GUI.
color	string	No	Name of a color to assign to this node. Possible values are “red”, “blue”, “green”, “orange”, and “gray”.
icon	string	No	Name of an icon to use for this node. See here for a list of all possible icon values.
value	any	No	The value for this node, if it is an I/O type.
readonly	bool	No	True if this node’s value field should only be read only. This will effect the API permissions and also the GUI’s display widget.
units	string	No	The unit of measure for the value field. (“V”, “A”, “mA”, etc.) Used by the GUI and APIs to help clarify to the user what numerical value means.

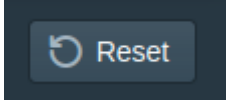
Field	Type	Required	Notes
format	string	No	The formatting specifier of how to format the value number. See here for the documentation for this format specifier language.
store	string	No	Set to “hourmeter” to save a rapidly changing readonly IO. Set to “config” to save and restore a seldom changed writable IO.

Possible Node Types

Name	Description
node	Just a node by itself, no special behavior of functions. Useful for organizing other nodes into groups in a logical way.

Possible IO Types

Name	Description	GUI Element
analog_io	A double floating point IO, used for any numerical type data you may need.	
digital_io	A boolean IO, used for any digital signals, flags, check-boxes, or status LEDs.	
string_io	A string IO, for any length string. Useful for configuration values, states, IDs, file names,	

Name	Description	GUI Element
button_io	Like a digital_io except, the GUI element is now a button that will set the digital to true when the button is pressed.	

A Simple Example

Lets say you have a new `T1` and you'd like the T1 to keep track of a special calculated value. What you want to do is add a new `analog_io` node to the T1 XML.

Your default T1 XML will look something like the following.

```
<root>
  <t1/>
  <epics_server/>
</root>
```

By default your `t1` node is configured for the completely standard set up. What we need to do is add out new node bellow it.

```
<root>
  <t1>
    <analog_io name="my_io" label="My IO" value="1.234" />
  </t1>
  <epics_server/>
</root>
```

As you can see, we simply “expand” the `t1` node and place our new `analog_io` node inside.

4 IGX Standard IO Interfaces

4.1 IGX Dose Controller IO Interface



The IGX Dose Controller features are in closed Alpha testing. This interface is subject to significant changes and is unsuitable for clinical use at this time. Please contact Pyramid for further information.

4.1.1 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 (pC, nA, V, counts, etc.). 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](#) for more information about the fundamentals of how to interface with IO themselves.

Interface 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.

Name	Detail	Type
<code>progress</code>	The progress of the current dose delivery.	Read-only Analog
<code>start_button</code>	Start the dose controller.	Digital
<code>pause_button</code>	Pause the dose controller.	Digital

Name	Detail	Type
reset_button	Reset current dose delivery.	Digital
dose_prescription	The prescription in real dose units to target.	Analog
detector_prescription	The prescription based on detector units.	Analog
time_prescription	The prescription based on time units.	Analog
point_dose_prescription	The prescription in real dose units to target for current control point.	Analog
point_detector_prescription	The prescription based on detector units for current control point.	Analog
point_time_prescription	The prescription based on time units for current control point.	Analog
dose_accumulation	The accumulated dose during the session.	Read-only Analog
detector_accumulation	The accumulated detector during the session.	Read-only Analog
time_elapsed	The total elapsed time over the session.	Read-only Analog
time_actuated	The actuated time during the session.	Read-only Analog

Name	Detail	Type
<code>point_dose_accumulation</code>	The accumulated dose during the current control point.	Read-only Analog
<code>point_detector_accumulation</code>	The accumulated detector during the current control point.	Read-only Analog
<code>point_time_elapsed</code>	The elapsed time over the current control point.	Read-only Analog
<code>point_time_actuated</code>	The time during the current control point that the controller was actuated.	Read-only Analog
<code>dose_rate</code>	The current dose delivery rate.	Read-only Analog
<code>detector_rate</code>	The current detector delivery rate.	Read-only Analog
<code>maximum_time_elapsed</code>	Upper limit of time for the session.	Analog
<code>detector_rate_min</code>	Lower limit of detector rate while treating.	Analog
<code>detector_rate_max</code>	Upper limit of detector rate while treating.	Analog

4.1.2 Use Case Examples

Lets say you want to program a single delivery of 100pC of charge in no longer than 1 second. You would

1. Set `reset_button` to 1 and then 0 to reset the controller.
2. Set `detector_prescription` to 100pC.
3. Set `maximum_time_elapsed` to 1s.

4. Set `start_button` to 1 and then 0, starting the session.
5. Wait for the delivery to complete, you can optionally monitor the `progress` as a percentage, or `detector_accumulation` as an accumulated charge.

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](#) 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.

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

Name	Detail	Type
monitor_voltage_internal	The output voltage as measured by the internal sense circuit directly on the output. Units are in volts.	Read-only Analog
monitor_voltage_external	The detected voltage as measured by the external sense circuit. Units are in volts.	Read-only Analog

Module States

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

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
```