



DEWETRON



---

# **OXYGEN DATA STREAM PLUGIN TECHNICAL REFERENCE MANUAL**

**Version:** 1.7  
**Date:** 2024-07-09  
**Author:** Gunther Laure



DEWETRON

The information contained in this document is subject to change without notice.

DEWETRON GmbH (DEWETRON) shall not be liable for any errors contained in this document. DEWETRON MAKES NO WARRANTIES OF ANY KIND WITH REGARD TO THIS DOCUMENT, WHETHER EXPRESS OR IMPLIED. DEWETRON SPECIFICALLY DISCLAIMS THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

DEWETRON shall not be liable for any direct, indirect, special, incidental, or consequential damages, whether based on contract, tort, or any other legal theory, in connection with the furnishing of this document or the use of the information in this document.

## Technical Support

Please contact your local authorized DEWETRON representative first for any support and service questions.

For Asia and Europe, please contact:

### **DEWETRON GmbH**

Parking 4  
8074 Grambach  
AUSTRIA

Tel.: +43 316 3070  
Fax: +43 316 307090  
Email: [support@dewetron.com](mailto:support@dewetron.com)  
Web: <http://www.dewetron.com>

For America, please contact:

### **DEWETRON, Inc.**

PO Box 1460  
Charlestown, RI 02813  
U.S.A.

Tel.: +1 401 364 9464  
Toll-free: +1 877 431 5166  
Fax: +1 401 364 8565  
Email: [us.support@dewetron.com](mailto:us.support@dewetron.com)  
Web: <http://www.dewetron.us>

The telephone hotline is available Monday to Friday between 08:00 and 17:00 GST (GMT -5:00)

## Restricted Rights Legend:

Use Austrian law for duplication or disclosure.

### **DEWETRON GmbH**

Parking 4  
8074 Grambach  
AUSTRIA

## Printing History:

Please refer to the page bottom for printing version. Copyright © DEWETRON GmbH



DEWETRON

This document contains information which is protected by copyright. All rights are reserved. Reproduction, adaptation, or translation without prior written permission is prohibited, except as allowed under the copyright laws.

All trademarks and registered trademarks are acknowledged to be the property of their owners.

Before updating your software please contact DEWETRON. Use only original software from DEWETRON.

Please find further information at [www.dewetron.com](http://www.dewetron.com).



DEWETRON

## DOCUMENT HISTORY

<b>Document Type:</b>	Technical Reference Manual
<b>Document Name:</b>	OXYGEN DATA STREAM Plugin
<b>Document Owner:</b>	Michael Oberhofer

Date	Author	Changes	Rev.
2018-11-06	Gunther Laure	First revision of the document for V1.5	0.1
2018-11-08	Gunther Laure	Added Example explanation and source code.	0.2
2018-11-20	Gunther Laure	Timestamp frequency is double instead of uint64	0.3
2018-11-27	Thomas Kastner	More XML and sample layout description	0.4
2020-09-01	Michael Oberhofer	Added Graphical Sketches for Packet Types	1.0
2021-03-25	Thomas Kastner	Updated Sample Data Type for CAN packets	1.1
2021-03-26	Thomas Kastner	Increase version number to V1.6	1.2
2022-08-02	Thomas Kastner	Add SCPI TRIG command description + increase version number to 1.8	1.3
2024-06-28	Thomas Kastner	Add offline stream description + add SCPI INTERVAL command	1.4



# Table of Contents

1	INTRODUCTION .....	7
1.1	SCOPE .....	7
1.2	FEATURES.....	7
1.3	RELATED DOCUMENTS.....	7
2	OXYGEN SETUP .....	8
2.1	EVALUATION LICENSE LIMITATIONS.....	8
2.2	STREAMING .....	8
2.3	OFFLINE STREAMING .....	8
2.4	SCPI COMMANDS .....	9
2.4.1	:DSTREAM:ITEMs[<GRP>] <CHANNEL>[,<CHANNEL>[,...]] .....	10
2.4.2	:DSTREAM:ITEMs[<GRP>]? .....	10
2.4.3	:DSTREAM:PORT[<GRP>] <PORT> .....	11
2.4.4	:DSTREAM:PORT[<GRP>]? .....	11
2.4.5	:DSTREAM:INIT [<GRP>   ALL].....	11
2.4.6	:DSTREAM:START [<GRP>   ALL] .....	12
2.4.7	:DSTREAM:STOP [<GRP>   ALL].....	12
2.4.8	:DSTREAM:DELETE [<GRP>   ALL] .....	12
2.4.9	: DSTREAM:RESET .....	13
2.4.10	: DSTREAM:STATE[<GRP>]?.....	13
2.4.11	: DSTREAM:TRIG[<GRP>] <SWITCH>.....	14
2.4.12	: DSTREAM:TRIG[<GRP>]?.....	14
2.4.13	: DSTREAM:INTERVAL[<GRP>] <INTERVAL>.....	15
2.4.14	: DSTREAM:INTERVAL[<GRP>]? .....	15
2.4.15	: DSTREAM:REPLAY[<GRP>] <MODE>.....	15
2.4.16	: DSTREAM:REPLAY[<GRP>]? .....	16
3	PROTOCOL DEFINITION .....	17
3.1	STREAM EXAMPLES.....	17
3.1.1	STREAM START PACKAGE.....	17
3.1.2	STREAM DATA PACKAGE.....	17
3.1.3	STREAM END PACKAGE .....	17
3.2	SUB-PACKET TYPES.....	18
3.3	PACKET HEADER (PH) .....	18
3.4	PACKET INFO (PI) .....	19



DEWETRON

3.5	XML PACKET (XML) .....	20
3.6	PACKET FOOTER (PF) .....	21
3.7	SYNCHRONOUS CHANNEL WITH FIXED SAMPLE SIZE (D) .....	22
3.8	SYNCHRONOUS CHANNEL WITH VARIABLE SAMPLE SIZE (D) .....	23
3.9	ASYNCHRONOUS CHANNEL WITH FIXED SAMPLE SIZE (D) .....	24
3.10	ASYNCHRONOUS CHANNEL WITH VARIABLE SAMPLE SIZE (D) .....	24
3.11	SAMPLE DATA TYPES .....	25
4	EXAMPLES .....	27
4.1	SCPI COMMAND SEQUENCE .....	27
4.2	CLIENT IN C .....	28
4.2.1	SOURCE CODE .....	28
4.3	EXTENSIVE CLIENT IN C++ .....	36
4.4	CLIENT IN PYTHON .....	37

# 1 INTRODUCTION

This document describes the “OXYGEN Data Stream” plugin.

The intended audience of this document are instrument programmers who are responsible for writing socket-based programs to access OXYGEN measurement data using network sockets (TCPIP).

## 1.1 SCOPE

This document describes the OXYGEN data stream plugin, the used protocol and its SCPI control interface. It contains small programming examples explaining how to access the data stream.

## 1.2 FEATURES

- High speed data access
- Efficient raw data transfer
- Multi data stream support
- Multi network port support
- Configurable via SCPI



## 1.3 RELATED DOCUMENTS

Refer to following documents for more information:

- OXYGEN 3.3 Feature Manual. This document describes the operation of OXYGEN software and its components.
- OXYGEN Remote Control – SCPI. This document describes the SCPI interface to communicate with OXYGEN.
- Standards Commands for Programmable Instruments (SCPI), Volume 1-4, Version 1999.0 May 1999, SCPI Consortium.
- Standard digital interface for programmable instrumentation – Part 2: Codes, formats, protocols and common commands, IEC 60488-2 First Edition 2004-5, IEEE.



DEWETRON

## 2 OXYGEN SETUP

The OXYGEN data stream plugin is part of the standard installer. To be able to use the plugin an OXYGEN OXY-OPT-DATASTREAM license is needed.

Without this license option it is not possible to start the plugin using the described SCPI interface (for testing purpose see Evaluation License Limitations).

### 2.1 EVALUATION LICENSE LIMITATIONS

It is possible to test the plugin in Evaluation mode, but streaming is limited to 5 minutes. After 5 minutes, any started streaming group will enter an ERROR state and the whole data streaming system needs to be reset.

### 2.2 STREAMING

After connecting a client to a registered data stream port, the OXYGEN data stream plugin initially sends a welcome message to the newly connected client, identifying itself as data stream plugin along with the actual protocol version.

Data stream welcome message
OXYGEN DATA STREAM PLUGIN V1.8

If the acquisition is restarted while a streaming group is active, this group is stopped and restarted, in case all used channels are still present and valid. If this is not the case, the according streaming group is deleted.

### 2.3 OFFLINE STREAMING

Initializing a stream group, when a DMD file is opened in Oxygen will give the possibility to stream channels from the DMD file. Offline streams can be configured and started in the same way, as streams during acquisition.

After the end of the file has been reached, the stream is automatically stopped.

The welcome message will indicate if it is an offline stream.

Data stream welcome message
OXYGEN <b>OFFLINE</b> DATA STREAM PLUGIN V1.8



## 2.4 SCPI COMMANDS

The SCPI DSTREAM subsystem of Oxygen allows the user to configure and control fast data streaming via a TCP network protocol. This documentation describes the SCPI commands to control the subsystem.

The following SCPI commands can be used to configure and control one or more streaming groups (distinguished via a GRP number). Each group has its own list of channels and can have a unique TCP port where data is provided. Each group can be configured and started individually. Some commands provide an ALL option to start/stop all configured groups.

Each streaming group has its own internal state. It can be queried using the STATE? query. The following list describes the allowed operations in each state:

- INVALID:** An invalid state means the streaming group does not exist, it will be automatically created when calling configuration commands such as ITEMS and PORT
- CONFIG:** In the configuration state, ITEMS and PORT commands can be used to configure the group. The INIT command will initialize the subsystem.
- INITIALIZED:** The streaming group is initialized (e.g. has an open TCP port) and is waiting for connections and the START command. If a configuration command such as ITEMS or PORT is executed, the state is set back to CONFIG.
- RUNNING:** The streaming group is actively sending data. Call STOP to enter the INITIALIZED state again.
- ERROR:** If one of the channels to register is unused, the whole dataset will be discarded and the DSTREAM system will be in an error state. Use :DStream:STATE? to get more details. It can be reset using the RESet command.

Refer to the following table for the symbols used to describe the syntax of commands and queries.

Symbol	Meaning
< >	A defined element
	Exclusive OR
[ ]	Optional; can be omitted
...	Previous elements can be repeated



### 2.4.1 :DSTREAM:ITEMS[<GRP>] <CHANNEL>[,<CHANNEL>[,...]]

<b>Syntax</b>	:DSTREAM:ITEMS[<GRP>] <CHANNEL>[,<CHANNEL>[,...]]			
<b>Description</b>	Defines the ordered list of requested channels for one stream group GRP			
<b>Parameter</b>	<b>Name</b>	<b>Type</b>	<b>Range</b>	<b>Default</b>
	GRP	NR1	Group number >= 1	1
	CHANNEL	ASCII string	Oxygen Channel Name	None
<b>Explanation</b>	<p>The user can specify a list of multiple channels.</p> <p>If one channel does not exist, is invalid or disabled, no channel is set for the streaming group (see system error log for more details). In addition, the following channel types are not supported: Digital channels, video channels and Rosette group channels.</p>			
<b>Example</b>	<p>Set two channels for stream group 2: AI 1/1 and AI 1/2</p> <p>→ :DST:ITEMs2 "AI 1/1", "AI 1/2"</p>			

### 2.4.2 :DSTREAM:ITEMs[<GRP>]?

<b>Syntax</b>	:DSTREAM:ITEMs[<GRP>]?			
<b>Description</b>	Returns the current list of requested channels for one channel group GRP			
<b>Parameter</b>	<b>Name</b>	<b>Type</b>	<b>Range</b>	<b>Default</b>
	GRP	NR1	Group number >= 1	1
<b>Explanation</b>	Returns NONE if no items are set			
<b>Example</b>	<p>Query the items of stream group 2:</p> <p>← :DST:ITEMs2?</p> <p>← "AI 1/1", "AI 1/2"</p>			



### 2.4.3 :DSTREAM:PORT[<GRP>] <PORT>

<b>Syntax</b>	:DSTREAM:PORT[ <GRP> ] <PORT>			
<b>Description</b>	Configures the port number of the TCP server for a streaming group GRP			
<b>Parameter</b>	<b>Name</b>	<b>Type</b>	<b>Range</b>	<b>Default</b>
	GRP	NR1	Group number >= 1	1
	PORT	NR1	1 .. 65536	10003
<b>Explanation</b>	The port must be a valid TCP port. It must not be used by any other TCP server on the local system but can be shared by multiple streaming groups of the Oxygen instance.			
<b>Example</b>	Set the TCP port for stream group 2 to 10005: → :DST:PORT2 10005			

### 2.4.4 :DSTREAM:PORT[<GRP>]?

<b>Syntax</b>	:DSTREAM:PORT[ <GRP> ]?			
<b>Description</b>	Queries the currently configured TCP port			
<b>Parameter</b>	<b>Name</b>	<b>Type</b>	<b>Range</b>	<b>Default</b>
	GRP	NR1	Group number >= 1	1
<b>Explanation</b>	Returns the NR1 numeric value of the configured TCP port			
<b>Example</b>	→ :DST:PORT2? ← 10005			

### 2.4.5 :DSTREAM:INIT [<GRP> | ALL]

<b>Syntax</b>	:DSTREAM:INIT [ <GRP>   ALL ]			
<b>Description</b>	Initializes one or all data streaming groups and opens the TCP port			
<b>Parameter</b>	<b>Name</b>	<b>Type</b>	<b>Range</b>	<b>Default</b>
	GRP	NR1	Group number >= 1	1
	ALL	Literal	Start all groups	None
<b>Explanation</b>	The server will listen for new connection on the configured port			
<b>Example</b>	Initializes the streaming group 1: → :DST:INIT 1			



#### 2.4.6 :DSTREAM:START [<GRP> | ALL]

<b>Syntax</b>	:DSTREAM:START [<GRP>   ALL]			
<b>Description</b>	Starts streaming of one or all data streaming groups			
<b>Parameter</b>	<b>Name</b>	<b>Type</b>	<b>Range</b>	<b>Default</b>
	GRP	NR1	Group number >= 1	1
	ALL	Literal	Start all groups	None
<b>Explanation</b>	Only possible after calling INIT for the specified group(s).			
<b>Example</b>	Start streaming for all configured streaming groups: → :DST:START ALL			

#### 2.4.7 :DSTREAM:STOP [<GRP> | ALL]

<b>Syntax</b>	:DSTREAM:STOP [<GRP>   ALL]			
<b>Description</b>	Stops streaming of one or all started data streaming groups			
<b>Parameter</b>	<b>Name</b>	<b>Type</b>	<b>Range</b>	<b>Default</b>
	GRP	NR1	Group number >= 1	1
	ALL	Literal	Start all groups	None
<b>Explanation</b>	Only possible after calling START for the specified group(s). After stopping, the streaming group is in INITIALIZED state.			
<b>Example</b>	Stop streaming for all configured streaming groups: → :DST:STOP ALL			

#### 2.4.8 :DSTREAM:DELETE [<GRP> | ALL]

<b>Syntax</b>	:DSTREAM:DELETE [<GRP>   ALL]			
<b>Description</b>	Deletes a configured streaming group			
<b>Parameter</b>	<b>Name</b>	<b>Type</b>	<b>Range</b>	<b>Default</b>
	GRP	NR1	Group number >= 1	1
	ALL	Literal	Start all groups	None
<b>Explanation</b>	The streaming group must be in config state. All ITEMS and port settings are deleted.			
<b>Example</b>	Delete streaming group 1: → :DST:DEL 1			

**2.4.9 : DSTREAM:RESEt**

<b>Syntax</b>	: DSTREAM:RESEt
<b>Description</b>	Resets the data streaming subsystem
<b>Parameter</b>	None
<b>Explanation</b>	Stops the server if started, and resets the configuration to its default settings
<b>Example</b>	→ :DST:RES

**2.4.10 : DSTREAM:STATE[<GRP>]?**

<b>Syntax</b>	: DSTREAM:STATE[<GRP>]?			
<b>Description</b>	Queries the internal state of the data streaming subsystem			
<b>Parameter</b>	<b>Name</b>	<b>Type</b>	<b>Range</b>	<b>Default</b>
	<GRP>	NR1	>= 1	1
<b>Explanation</b>	<p>Possible states are:</p> <p>INVALID: Streaming group does not exist</p> <p>CONFIG: Configuration state (allows settings of ITEMS and PORT)</p> <p>INITIALIZED: The streaming group has already been initialized</p> <p>RUNNING: The server is started and cannot be configured anymore</p> <p>ERROR: An error has occurred; the user needs to call RESEt</p> <p>UNLICENSED: A valid license for the data streaming subsystem was not found</p>			
<b>Example</b>	<p>Query the state of the streaming group 1 (default value):</p> <p>→ :DST:STATE?</p> <p>← CONFIG</p>			

**2.4.11 : DSTREAM:TRIG[<GRP>] <SWITCH>**

<b>Syntax</b>	: DSTREAM:TRIG[<GRP>] <SWITCH>			
<b>Description</b>	Configure a streaming group to send data only when a trigger is active			
<b>Parameter</b>	<b>Name</b>	<b>Type</b>	<b>Range</b>	<b>Default</b>
	<GRP>	NR1	>= 1	1
	<SWITCH>	ASCII string	ON/OFF	OFF
<b>Explanation</b>	When this property is enabled for a streaming group, data will only be streamed during a triggered measurement, while any trigger is active.			
<b>Example</b>	→ :DST:TRIG1 ON			

**2.4.12 : DSTREAM:TRIG[<GRP>]?**

<b>Syntax</b>	: DSTREAM:TRIG[<GRP>]?			
<b>Description</b>	Query the triggered property for a given streaming group			
<b>Parameter</b>	<b>Name</b>	<b>Type</b>	<b>Range</b>	<b>Default</b>
	<GRP>	NR1	>= 1	1
<b>Explanation</b>	Returns the current state of the triggered property for a stream group, possible values are ON (triggered mode is active), or OFF (data is streamed continuously)			
<b>Example</b>	→ :DST:TRIG1?			
	→ ON			

**2.4.13 : DSTREAM:INTERVAL[<GRP>] <INTERVAL>**

<b>Syntax</b>	: DSTREAM:INTERVAL[ <GRP>] <INTERVAL>			
<b>Description</b>	Configure a streaming group to send data blocks with a given time interval			
<b>Parameter</b>	<b>Name</b>	<b>Type</b>	<b>Range</b>	<b>Default</b>
	<GRP>	NR1	>= 1	1
	<INTERVAL>	NR1	100 .. 1000	100
<b>Explanation</b>	When this property is set for a streaming group, data will be sent in blocks of the given time interval.			
<b>Example</b>	→ :DST:INTERVAL1 500			

**2.4.14 : DSTREAM:INTERVAL[<GRP>]?**

<b>Syntax</b>	: DSTREAM:INTERVAL[ <GRP>]?			
<b>Description</b>	Query the interval property for a given streaming group			
<b>Parameter</b>	<b>Name</b>	<b>Type</b>	<b>Range</b>	<b>Default</b>
	<GRP>	NR1	>= 1	1
<b>Explanation</b>	Returns the current value of the interval property for a stream group, possible values are between 100 and 1000 milliseconds.			
<b>Example</b>	→ :DST:INTERVAL1?			
	→ 500			

**2.4.15 : DSTREAM:REPLAY[<GRP>] <MODE>**

<b>Syntax</b>	: DSTREAM:REPLAY[ <GRP>] <MODE>			
<b>Description</b>	Configure an offline streaming group to send data blocks in live or bulk mode			
<b>Parameter</b>	<b>Name</b>	<b>Type</b>	<b>Range</b>	<b>Default</b>
	<GRP>	NR1	>= 1	1
	<MODE>	ASCII string	BULK, LIVE	BULK
<b>Explanation</b>	When this property is set for an offline streaming group, data will be sent in live speed, or as fast as possible when set to BULK mode.			
<b>Example</b>	→ :DST:REPLAY1 LIVE			

#### 2.4.16 : DSTREAM:REPLAY[<GRP>]?

<b>Syntax</b>	: DSTREAM:INTERVAL[ <GRP> ]?			
<b>Description</b>	Query the replay property for a given offline streaming group			
<b>Parameter</b>	<b>Name</b>	<b>Type</b>	<b>Range</b>	<b>Default</b>
	<GRP>	NR1	>= 1	1
<b>Explanation</b>	Returns the current value of the replay property for an offline stream group, possible values are BULK and LIVE.			
<b>Example</b>	→ :DST:REPLAY1? → LIVE			





## 3 PROTOCOL DEFINITION

This is the definition of the OXYGEN Data Stream Protocol Version 1.6.

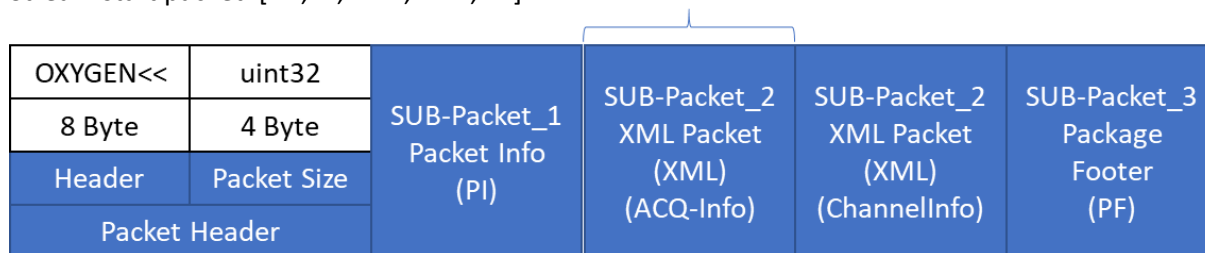
A complete stream packet consists of a number of defined sub-packets. All sub-packets follow the same basic format (sub-packet size, type) so it is possible to iterate a complete stream packet with this information only. Only the packet header (PH) does not follow this pattern.

### 3.1 STREAM EXAMPLES

There are 3 different Stream Packets, depending on the state of the data stream.

#### 3.1.1 STREAM START PACKAGE

Stream start packet: [PH, PI, XML, XML, PF]

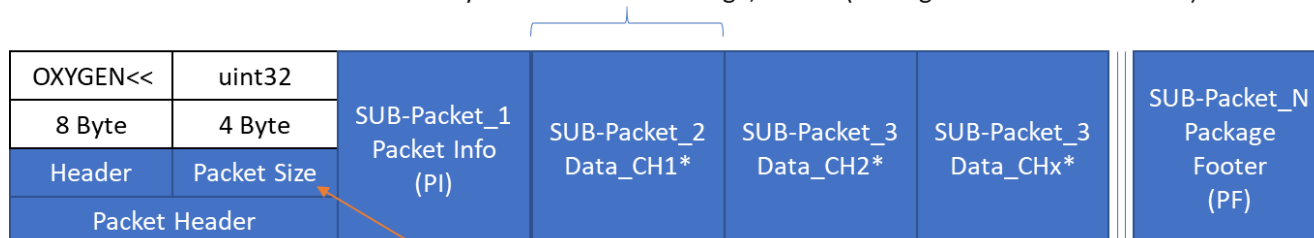


Packet Size

#### 3.1.2 STREAM DATA PACKAGE

Stream data packet: [PH, PI, D, D, ... , PF]

SUB-Packet Size = first 4 Bytes of each Sub-Package, uint32 (see e.g. Section 3.7 in Techref)



Packet Size

#### 3.1.3 STREAM END PACKAGE

Stream end packet: [PH, PI, PF]



DEWETRON

### 3.2 SUB-PACKET TYPES

<i>Sub Packet Types</i>	<i>Description</i>	<i>Comment</i>
0x00000001	Packet Info (PI)	Stream packet detail data
0x00000002	XML Packet (XML)	Generic configuration data
0x00000003	Sync channel (fixed sample size)	Sync sample data
0x00000004	Sync channel (variable sample size)	Sync sample data
0x00000005	Async channel (fixed sample size)	Async sample data
0x00000006	Async channel (variable sample size)	Async sample data
0x00000007	Packet Footer (PF)	Checksum and packet end string

### 3.3 PACKET HEADER (PH)

The header packet contains a defined start string and a size field. The packet header will never change for newer protocols (immutable for future versions).

<i>Offset (bytes)</i>	<i>Length (bytes)</i>	<i>Data Type</i>	<i>Value</i>	<i>Description</i>	<i>Comment</i>
0	8	string	"OXYGEN<<"	Start packet token	Start string.
8	4	uint32		Packet size	Size of the complete packet starting from offset 0.
12					Header size



### 3.4 PACKET INFO (PI)

Packet info follows the Packet Header (PH) sub-packet. This packet holds the essential information for decoding following packets. It holds the unique stream id, the stream specific sequence number, the number of sub-packets following this stream.

Special fields are:

- Stream Status: Marking the first, last packet of a stream.
- Random Seed: Can be ignored for now.

Offset (bytes)	Length (bytes)	Data Type	Value	Description	Comment
0	4	uint32	32	Packet info size	Size of this subpacket from offset 0.
4	4	uint32	0x00000001	Packet Info	
8	4	uint32	0x01050000	Protocol Version	Version number encoded in upper 16bit
12	4	uint32		Stream ID	Unique ID to be able to differentiate streams.
16	4	uint32		Sequence Number	Counting stream packets.
20	4	uint32		Stream Status	Mark first, last, error packet
24	4	uint32		Random Seed	Random number to increase entropy.
28	4	uint32		Number of following sub-packets	Packet info not included in count.
32					



DEWETRON

## Stream Status

<i>Stream Status</i>	<i>Description</i>	<i>Comment</i>
0x00000001	First packet of a stream	First packet after stream start or client connect
0x00000002	Last packet of a stream	Last packet after stopping the stream or acquisition
0x00000000	Normal in-between data packet	
0x10000000	Error packet	Currently not used

## 3.5 XML PACKET (XML)

Stream channel information according the acquisition start and channel scaling information is transferred using XML Packet sub-packets.

As of protocol version 1.6 only the first packet of a stream will contain XML packets.

<i>Offset (bytes)</i>	<i>Length (bytes)</i>	<i>Data Type</i>	<i>Value</i>	<i>Description</i>	<i>Comment</i>
0	4	uint32	32	XML packet size	Size of this subpacket from offset 0.
4	4	uint32	0x00000002	Packet Info: XML Packet	see Sub-Packet Types
8	sizeof(XML)	XML string		Channel Information, ACQ information	valid XML document (size = XML Packet size – 8)
XML packet size					



DEWETRON

Currently support XML documents are:

### Acquisition Information

```
<?xml version="1.0"?>
<AcquisitionInfo>
  <AcquisitionStart>
    <Timestamp format = "ISO8061">2022-12-20T10:14:41.136+01</Timestamp>
    <Timestamp format = "nanoseconds" utc_offset_seconds = "3600">1671527681136000000</Timestamp>
  </AcquisitionStart>
</AcquisitionInfo>
```

### Channel Information

```
<?xml version="1.0"?>
<ChannelInfo>
  <Channel name="AI 1/1" unit="V">
    <LinearScaling factor="4.65661e-08" offset="0"/>
  </Channel>
  <Channel name="CNT 1/1" unit="V">
    <NoScaling />
  </Channel>
</ChannelInfo>
```

## 3.6 PACKET FOOTER (PF)

Offset (bytes)	Length (bytes)	Data Type	Value	Description	Comment
0	4	uint32	32	Footer size	Size of this subpacket from offset 0.
4	4	uint32	0x00000007	Packet Info: Packet Footer	see Sub-Packet Types
8	4	uint32		Checksum	CRC32 for header and all sub-packets without this footer packet
12	8	string	">>OXYGEN"	End packet token	End string
20					



DEWETRON

### 3.7 SYNCHRONOUS CHANNEL WITH FIXED SAMPLE SIZE (D)

The format for one sample is: SAMPLE can be any fixed size entry from “Sample Data Types”.

<i>Offset (bytes)</i>	<i>Length (bytes)</i>	<i>Data Type</i>	<i>Value</i>	<i>Description</i>	<i>Comment</i>
0	4	uint32	32	Packet size	Size of this subpacket from offset 0.
4	4	uint32	0x00000003	Packet Info: Sync channel fixed sample size	see Sub-Packet Types
8	4	uint32		Sample data type	see “Sample Data Types” table
12	4	uint32		Dimension	1 for scalar channels
16	4	uint32		Number of samples	Number of samples
20	8	uint64		Timestamp	Timestamp of first sample
28	8	double		Timebase Frequency	equals sample rate
36	Packet size - 36			Data samples	array of samples
Packet size					



DEWETRON

### 3.8 SYNCHRONOUS CHANNEL WITH VARIABLE SAMPLE SIZE (D)

The format for one sample can change for every sample. The sample format is [sample size, sample value].  
The sample size is stored as uint32 value.

<i>Offset (bytes)</i>	<i>Length (bytes)</i>	<i>Data Type</i>	<i>Value</i>	<i>Description</i>	<i>Comment</i>
0	4	uint32	32	Packet size	Size of this subpacket from offset 0.
4	4	uint32	0x00000004	Packet Info: Sync channel variable sample size	see Sub-Packet Types
8	4	uint32		Sample data type	see “Sample Data Types” table
12	4	uint32		Dimension	1 for scalar channels
16	4	uint32		Number of samples	Number of samples
20	8	uint64		Timestamp	Timestamp of first sample
28	8	double		Timebase Frequency	equals sample rate
36	Packet size - 36			Data samples	array of samples
Packet size					



DEWETRON

### 3.9 ASYNCHRONOUS CHANNEL WITH FIXED SAMPLE SIZE (D)

Asynchronous samples have a timestamp for each sample. The timestamp is 8 bytes in ticks from acquisition start based on the channel's timebase frequency.

The format for one sample is fixed. The sample format is [timestamp, sample value].

The timestamp is stored as uint64 value.

Offset (bytes)	Length (bytes)	Data Type	Value	Description	Comment
0	4	uint32	32	Packet size	Size of this subpacket from offset 0.
4	4	uint32	0x00000005	Packet Info: Async channel fixed sample size	see Sub-Packet Types
8	4	uint32		Sample data type	see "Sample Data Types" table
12	4	uint32		Dimension	1 for scalar channels
16	4	uint32		Number of samples	Number of samples
20	8	double		Timebase Frequency	equals sample rate
28	Packet size - 28			Data samples	array of samples
Packet size					

### 3.10 ASYNCHRONOUS CHANNEL WITH VARIABLE SAMPLE SIZE (D)

Asynchronous samples have a timestamp for each sample. The timestamp is 8 bytes in ticks from acquisition start based on the channel's timebase frequency.

The data format for one sample can change for every sample. The sample format is [timestamp, sample size, sample value].

The timestamp is stored as uint64 value.

The sample size is stored as uint32 value.





DEWETRON

<i>Offset (bytes)</i>	<i>Length (bytes)</i>	<i>Data Type</i>	<i>Value</i>	<i>Description</i>	<i>Comment</i>
0	4	uint32	32	Channel packet size	Size of this subpacket from offset 0.
4	4	uint32	0x00000006	Packet Info: Async channel variable sample size	see Sub-Packet Types
8	4	uint32		Sample data type	see "Sample Data Types" table
12	4	uint32		Dimension	1 for scalar channels
16	4	uint32		Number of samples	Number of samples
20	8	double		Timebase Frequency	equals sample rate
28	Packet size - 28			Data samples	array of samples
Packet size					

### 3.11 SAMPLE DATA TYPES

<i>Sample Data Types</i>	<i>Data Type Name</i>	<i>Size (bytes)</i>
0	sint8	1
1	uint8	1
2	sint16	2
3	uint16	2
4	sint24	3



DEWETRON

5	uint24	3
6	sint32	4
7	uint32	4
8	sint64	8
9	uint64	8
10	float	4
11	double	8
12	complex float	8
13	complex double	16
14	string	variable
15	binary	variable
16	CAN	variable (5 bytes ID + 4 bytes sample size + message)



DEWETRON

## 4 EXAMPLES

The plugin comes with a number of examples for reading data streams. The examples are written in C, C++ and Python. We tried to write the example operating system independent and tested them at least on Windows and Linux.

The example source code is free to reuse.

### 4.1 SCPI COMMAND SEQUENCE

This SCPI command sequence is used to configure and start data streaming. It must be executed for the client examples to be able to run. Text commands are sent via a TCP connection to the Oxygen SCPI server (port number 10001). Telnet provides an easy method to execute SCPI commands in Oxygen, but any other TCP client program or code is suitable.

```
user:~$ telnet <ip-address> 10001
```

The command sequence to verify if the connection is a valid Oxygen connection, one can query the system information via the following command:

```
Send:      *IDN?
Receive:   *IDN DEWETRON,OXYGEN,0,3.4.0
```

In order to set up data streaming for the examples, execute the following commands (note that these commands are only used in their most simple form without streaming group indices):

Configure Oxygen to listen on port 5555 and provide a stream for channels 'AI 1/1' and 'AI 1/2':

```
Send:      :DST:RESET
Send:      :DST:PORT 5555
Send:      :DST:ITEMS 'AI 1/1', 'AI 1/2'
Send:      :DST:INIT
```

Now it is possible to connect to port 5555 using the client program. After the client program has been connected, data streaming can be started using:

```
Send:      :DST:START
```

When data streaming is no longer needed, shutdown the data stream subsystem using the following command:

```
Send:      :DST:STOP
Send:      :DST:DELETE
```



DEWETRON

## 4.2 CLIENT IN C

The example program `dt_client_c` (C) is a platform independent client that is able to connect to a OXYGEN instance. It reads `DataStream` packets and processes all the packets meta information.

The client implements Data Stream Protocol V1.8.

The example uses cmake for generating Makefiles or a Visual Studio Solution (<https://cmake.org>).

Features shown:

- Connecting to a configured OXYGEN Data Stream plugin using sockets
- Processing the welcome message
- Listening for incoming stream packets
- Processing of stream sub-packets
- Print sub-packet information (omitting sample values)
- Automatically stops listening on stream end.

Files:

- `inc/dt_stream_packet_c.h`
- `src/dt_client_c.c`
- `CMakeLists.txt`

Only essential parts of the example source code are shown here. Macros and includes for platform independence have been removed.

The full client example is part of the Oxygen installation package.

### 4.2.1 SOURCE CODE

```
#include "dt_stream_packet_c.h"
```

```
/**
```

```
 * Connect to a OXYGEN data stream plugin, read and
 * process welcome message.
 * @param dt_server
 * @param port
 * @return socket
 */
```

```
socket_t connectTo(const char* dt_server, uint32_t port);
```

```
/**
```

```
 * Read a packet.
 * @param sock to read from
 * @return 0 if successful
 */
```



DEWETRON

```
int32_t readPacket(socket_t sock);
```

```
int32_t processHeader(const uint8_t* buffer, uint32_t packet_size);
```

```
int32_t processPacket(const uint8_t* buffer, uint32_t buffer_size);
```

```
const uint8_t* readData(const uint8_t* pos, void* target, ssize_t target_size);
```

```
int32_t processPacketInfo(const uint8_t* buffer, uint32_t buffer_size);
```

```
void processXmlConfig(const uint8_t* buffer, uint32_t buffer_size);
```

```
void processSyncFixed(const uint8_t* buffer, uint32_t buffer_size);
```

```
void processSyncVariable(const uint8_t* buffer, uint32_t buffer_size);
```

```
void processAsyncFixed(const uint8_t* buffer, uint32_t buffer_size);
```

```
void processAsyncVariable(const uint8_t* buffer, uint32_t buffer_size);
```

```
void processFooter(const uint8_t* buffer, uint32_t buffer_size);
```

```
/**
```

```
 * dt_client_c main
```

```
 * Implements: OXYGEN DATA TRANSFER Protocol 1.6
```

```
 * default address: 127.0.0.1
```

```
 * default port: 5555
```

```
 *
```

```
 * Example to access OXYGEN: dt_client_c 192.168.0.155 5555
```

```
 */
```

```
int main(int argc, char* argv[])
```

```
{
```

```
    char* address;
```

```
    uint32_t port;
```

```
    socket_t sock;
```

```
    switch(argc)
```

```
    {
```

```
    case 1:
```

```
        address = "127.0.0.1";
```

```
        port = 5555;
```

```
        break;
```

```
    case 3:
```

```
        address = argv[1];
```

```
        port = atoi(argv[2]);
```

```
        break;
```

```
    }
```

```
    sock = connectTo(address, port);
```

```
    if (sock < 0)
```

```
    {
```



DEWETRON

```
    fprintf(stderr, "connectTo %s:%d failed\n", address, port);
    return -1;
}

while(readPacket(sock) > 0)
{
    // nothing to do
}

return 0;
}

socket_t connectTo(const char* dt_server, uint32_t port)
{
    char welcome_buffer[DT_WELCOME_MSG_SIZE];
    struct sockaddr_in serv_addr;
    socket_t sock = 0;

    sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock < 0) return -1;

    memset(&serv_addr, 0x00, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(port);

    if (inet_pton(AF_INET, dt_server, &serv_addr.sin_addr) <= 0)
    {
        fprintf(stderr, "Invalid address\n");
        return -1;
    }

    if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0)
    {
        fprintf(stderr, "Connection to %s:%d failed\n", dt_server, port);
        return -1;
    }

    // after successful the client has to read the connection welcome message.
    memset(welcome_buffer, 0, DT_WELCOME_MSG_SIZE);
    ssize_t bc = recv(sock, welcome_buffer, DT_WELCOME_MSG_SIZE, 0);
    if (bc == 0)
    {
        fprintf(stderr, "Could not read welcome message");
        return -1;
    }
}
```



DEWETRON

```
printf("Data stream product name: %s\n", welcome_buffer);

return sock;
}

int32_t readPacket(socket_t sock)
{
    uint8_t packet_header_buffer[DT_PACKET_HEADER_SIZE];
    uint8_t* packet_data = 0;
    uint32_t packet_size = 0;
    int32_t ret = 1;
    int32_t bc = recv(sock, packet_header_buffer, DT_PACKET_HEADER_SIZE, 0);

    if (DT_PACKET_HEADER_SIZE != bc)
    {
        fprintf(stderr, "Could not read header\n");
        return -1;
    }

    if (processHeader(packet_header_buffer, &packet_size) < 0)
    {
        fprintf(stderr, "Could not process packet header\n");
        return -1;
    }

    // read rest of the packet
    packet_size = packet_size - DT_PACKET_HEADER_SIZE;
    packet_data = malloc(packet_size);
    bc = recv(sock, packet_data, packet_size, MSG_WAITALL);
    if (packet_size != bc)
    {
        fprintf(stderr, "Could not read all packet data\n");
        free(packet_data);
        return -1;
    }

    if ((ret = processPacket(packet_data, packet_size)) < 0)
    {
        fprintf(stderr, "Could not process packet\n");
        free(packet_data);
        return -1;
    }

    free(packet_data);
}
```



```
    return ret;
}

int32_t processHeader(const uint8_t* buffer, uint32_t* packet_size)
{
    char start_token[DT_START_TOKEN_SIZE];
    const uint8_t* pos = NULL;

    pos = readData(buffer, start_token, DT_START_TOKEN_SIZE);
    if (0 != strncmp(DT_START_TOKEN, start_token, DT_START_TOKEN_SIZE))
    {
        fprintf(stderr, "Invalid packet start token\n");
        return -1;
    }

    // read the packet size
    pos = readData(pos, packet_size, sizeof(uint32_t));

    return 0;
}

int32_t processPacket(const uint8_t* buffer, uint32_t buffer_size)
{
    const uint8_t* pos = buffer;
    const uint8_t* end = buffer + buffer_size;
    const uint8_t* start_pos = buffer;
    uint32_t packet_size = 0;
    uint32_t packet_type = 0;
    uint32_t hit_footer = 0;
    int32_t ret = 1;

    while ((pos < end) && (!hit_footer))
    {
        start_pos = pos;

        pos = readData(pos, &packet_size, sizeof(packet_size));
        pos = readData(pos, &packet_type, sizeof(packet_type));

        switch(packet_type)
        {
            case SBT_PACKET_INFO:    ret = processPacketInfo(pos, packet_size); break;
            case SBT_XML_CONFIG:    processXmlConfig(pos, packet_size); break;
            case SBT_SYNC_FIXED:    processSyncFixed(pos, packet_size); break;
            case SBT_SYNC_VARIABLE: processSyncVariable(pos, packet_size); break;
        }
    }
}
```





DEWETRON

```
case SBT_ASYNC_FIXED:    processAsyncFixed(pos, packet_size); break;
case SBT_ASYNC_VARIABLE: processAsyncVariable(pos, packet_size); break;
case SBT_PACKET_FOOTER:
    processFooter(pos, packet_size);
    hit_footer = 1; // ensure to end the loop after footer processing
    break;
default:
    fprintf(stderr, "Unsupported SubPacketType: %d (size: %d)\n", packet_type, packet_size);
    break;
}

// iterate to next subpacket
pos = start_pos + packet_size;
}

return ret;
}

const uint8_t* readData(const uint8_t* pos, void* target, ssize_t target_size)
{
    memcpy(target, &(*pos), target_size);
    pos += target_size;
    return pos;
}

int32_t processPacketInfo(const uint8_t* buffer, uint32_t buffer_size)
{
    DtPacketInfo sub_packet;
    const uint8_t* pos = buffer;

    pos = readData(pos, &sub_packet.protocol_version, sizeof(sub_packet.protocol_version));
    pos = readData(pos, &sub_packet.stream_id, sizeof(sub_packet.stream_id));
    pos = readData(pos, &sub_packet.sequence_number, sizeof(sub_packet.sequence_number));
    pos = readData(pos, &sub_packet.stream_status, sizeof(sub_packet.stream_status));
    pos = readData(pos, &sub_packet.seed, sizeof(sub_packet.seed));
    pos = readData(pos, &sub_packet.number_of_subpackets, sizeof(sub_packet.number_of_subpackets));

    fprintf(stdout, "PacketInfo:\n");
    fprintf(stdout, " Version:      %0x\n", sub_packet.protocol_version);
    fprintf(stdout, " Stream ID:    %d\n", sub_packet.stream_id);
    fprintf(stdout, " Seq Number:   %d\n", sub_packet.sequence_number);
    fprintf(stdout, " Stream status: %x\n", sub_packet.stream_status);
    fprintf(stdout, " Stream seed:  %x\n", sub_packet.seed);
    fprintf(stdout, " Num sub packets: %d\n", sub_packet.number_of_subpackets);
}
```



DEWETRON

```
    return (sub_packet.stream_status & ST_LAST_PACKET) == 0;
}

void processXmlConfig(const uint8_t* buffer, uint32_t buffer_size)
{
    DtXmlSubPacket sub_packet;
    const uint8_t* pos = buffer;

    sub_packet.xml_content_size = buffer_size - DT_PACKET_BASE_SIZE;
    sub_packet.xml_content = malloc(sub_packet.xml_content_size + 1);
    sub_packet.xml_content[sub_packet.xml_content_size] = 0;

    pos = readData(pos, sub_packet.xml_content, sub_packet.xml_content_size);

    fprintf(stdout, "XMLPacket:\n");
    fprintf(stdout, "  xml_content:    %s\n", sub_packet.xml_content);

    free(sub_packet.xml_content);
}

void processSyncFixed(const uint8_t* buffer, uint32_t buffer_size)
{
    DtChannelSyncFixed sub_packet;
    const uint8_t* pos = buffer;

    pos = readData(pos, &sub_packet.channel_data_type, sizeof(sub_packet.channel_data_type));
    pos = readData(pos, &sub_packet.channel_dimension, sizeof(sub_packet.channel_dimension));
    pos = readData(pos, &sub_packet.number_samples, sizeof(sub_packet.number_samples));
    pos = readData(pos, &sub_packet.timestamp, sizeof(sub_packet.timestamp));
    pos = readData(pos, &sub_packet.timebase_frequency, sizeof(sub_packet.timebase_frequency));

    fprintf(stdout, "DtChannelSyncFixed:\n");
    fprintf(stdout, "  channel_data_type: %s (%d)\n", getDtDataTypeName(sub_packet.channel_data_type),
sub_packet.channel_data_type);
    fprintf(stdout, "  channel_dimension: %d\n", sub_packet.channel_dimension);
    fprintf(stdout, "  number_samples:    %d\n", sub_packet.number_samples);
    fprintf(stdout, "  timestamp:         %lu\n", sub_packet.timestamp);
    fprintf(stdout, "  timebase_frequency %f\n", sub_packet.timebase_frequency);
}

void processSyncVariable(const uint8_t* buffer, uint32_t buffer_size)
{
    DtChannelSyncVariable sub_packet;
    const uint8_t* pos = buffer;
```



DEWETRON

```
pos = readData(pos, &sub_packet.channel_data_type, sizeof(sub_packet.channel_data_type));
pos = readData(pos, &sub_packet.channel_dimension, sizeof(sub_packet.channel_dimension));
pos = readData(pos, &sub_packet.number_samples, sizeof(sub_packet.number_samples));
pos = readData(pos, &sub_packet.timestamp, sizeof(sub_packet.timestamp));
pos = readData(pos, &sub_packet.timebase_frequency, sizeof(sub_packet.timebase_frequency));

fprintf(stdout, "DtChannelSyncVariable:\n");
fprintf(stdout, " channel_data_type: %s (%d)\n", getDtDataTypeName(sub_packet.channel_data_type),
sub_packet.channel_data_type);
fprintf(stdout, " channel_dimension: %d\n", sub_packet.channel_dimension);
fprintf(stdout, " number_samples: %d\n", sub_packet.number_samples);
fprintf(stdout, " timestamp: %lu\n", sub_packet.timestamp);
fprintf(stdout, " timebase_frequency %f\n", sub_packet.timebase_frequency);
}

void processAsyncFixed(const uint8_t* buffer, uint32_t buffer_size)
{
    DtChannelAsyncFixed sub_packet;
    const uint8_t* pos = buffer;

    pos = readData(pos, &sub_packet.channel_data_type, sizeof(sub_packet.channel_data_type));
    pos = readData(pos, &sub_packet.channel_dimension, sizeof(sub_packet.channel_dimension));
    pos = readData(pos, &sub_packet.number_samples, sizeof(sub_packet.number_samples));
    pos = readData(pos, &sub_packet.timebase_frequency, sizeof(sub_packet.timebase_frequency));

    fprintf(stdout, "DtChannelAsyncFixed:\n");
    fprintf(stdout, " channel_data_type: %s (%d)\n", getDtDataTypeName(sub_packet.channel_data_type),
sub_packet.channel_data_type);
    fprintf(stdout, " channel_dimension: %d\n", sub_packet.channel_dimension);
    fprintf(stdout, " number_samples: %d\n", sub_packet.number_samples);
    fprintf(stdout, " timebase_frequency %f\n", sub_packet.timebase_frequency);
}

void processAsyncVariable(const uint8_t* buffer, uint32_t buffer_size)
{
    DtChannelAsyncVariable sub_packet;
    const uint8_t* pos = buffer;

    pos = readData(pos, &sub_packet.channel_data_type, sizeof(sub_packet.channel_data_type));
    pos = readData(pos, &sub_packet.channel_dimension, sizeof(sub_packet.channel_dimension));
    pos = readData(pos, &sub_packet.number_samples, sizeof(sub_packet.number_samples));
    pos = readData(pos, &sub_packet.timebase_frequency, sizeof(sub_packet.timebase_frequency));

    fprintf(stdout, "DtChannelAsyncVariable:\n");
```



DEWETRON

```
    fprintf(stdout, " channel_data_type: %s (%d)\n", getDtDataTypeName(sub_packet.channel_data_type),
sub_packet.channel_data_type);
    fprintf(stdout, " channel_dimension: %d\n", sub_packet.channel_dimension);
    fprintf(stdout, " number_samples: %d\n", sub_packet.number_samples);
    fprintf(stdout, " timebase_frequency %f\n", sub_packet.timebase_frequency);
}

void processFooter(const uint8_t* buffer, uint32_t buffer_size)
{
    DtPacketFooter sub_packet;
    const uint8_t* pos = buffer;

    pos = readData(pos, &sub_packet.checksum, sizeof(sub_packet.checksum));

    fprintf(stdout, "PacketFooter:\n");
    fprintf(stdout, " Checksum: %0x\n", sub_packet.checksum);
}
```

### 4.3 EXTENSIVE CLIENT IN C++

The example program dt\_client (C++) is a platform independent client that is able to connect to a running OXYGEN application. It reads DataStream packets and processes all the packets meta information.

The client implements Data Stream Protocol V1.8.

The example uses cmake for generating Makefiles or Visual Studio Solution files (<https://cmake.org>).

In addition to example dt\_client\_c adds following features:

- Extensive commandline
- Processing acquisition start time
- Processing channel scaling information
- Printing channel sample values

Files:

- CMakeLists.txt
- inc/crc32.h
- inc/dt\_client.h
- inc/dt\_cmdline.h
- inc/dt\_log.h
- inc/dt\_stream\_packet.h
- src/dt\_client.cpp
- src/dt\_cmdline.cpp



DEWETRON

- `src/dt_log.cpp`
- `src/dt_stream_packet.cpp`
- `src/pugiconfig.hpp`
- `src/pugixml.cpp`
- `src/pugixml.hpp`

The full client example (including source code) is available for download.

#### **4.4 CLIENT IN PYTHON**

The client in Python is also available for download.