WebXi for SLM
Version 1.0

Name: WebXi for SLM version 1.0
Author: Helge Rasmussen
Protocol version: 1.0
Document version: 1.0

Revision History:

Initials	Version	Date	Comments
HELGERAS	1.0	23-12-2019	First Version for external use

Contents

Name: WebXi for SLM version 1.0
Author: Helge Rasmussen
Protocol version: 1.0
Document version: 1.0

1 Terminology

Term/Acronym/Abbreviation	Description
REST	REST (representational state transfer) is an approach for getting information content from a Web site by reading a designated Web page that contains an XML (Extensible Mark-up Language) or JSON (JavaScript Object Notation) file that describes and includes the desired content.
SLM	Sound Level Meter
Client	Software that communicates with a WebXi device such as a web browser or an app communicating with a type 2245 Sound Level Meter.
WebSocket protocol	A protocol for a bi-directional streaming documented in RFC 6455. See also http://www.websocket.org/

2 Introduction

This document describes version 1.0 of the WebXi protocol. This protocol is currently used in B&K's Sound Level Meter type 2245.

WebXi is a protocol for communication with devices developed by HBK. A device that supports the protocol is called a WebXi device. Such a device may be a hardware device, but can also be a pure software device, such as a service in the cloud.

The protocol is divided in two parts.
The first part is the ***Command Protocol***. This is used to examine the state, configure and command a WebXi-device

The other part is the ***Streaming Protocol***. This is used to receive or send one or more streams of data. This way of communication is useful for monitoring and other situations where continuous communication with the device is needed.

This document describes the concepts, syntax and details of the protocol, but does not describe device specific details. Look in the WebXi documentation for the device for this.

3 The Command Protocol

The Command Protocol is used to examine the state and configure and control a WebXi-device.

3.1 REST

The command protocol will be based on REST on top of HTTP; data is mostly encoded as JSON.
See "RESTful Web Services" by Leonard Richardson and Sam Ruby for details.

The general REST guidelines for using the 4 HTTP methods are:

- **GET**: Used to get information. There must be no side effects.
- **DELETE**: Delete something.
- **PUT**: Used to set/change information on existing resource.
- **POST**: Used to create a resource and possibly set values on it. Typically, the POST command will return the URL of the new resource, so it can be accessed and maybe deleted later.

3.2 Hierarchy

Data on the device is ordered in a tree of named "nodes". The node names are case insensitive.
A node is either a *leaf node*, which typically has a value, or a *branch node* which will have one or more other nodes as children.

It is possible to request the value of a node or a tree of nodes by using the "*recursive*" keyword. See 3.5 for details.

A node is specified by the normal Unix path notation. The top node is always named "`/WebXi`":
Here's an example:

```
/WebXi/Node1/Node2/Node3
```

Where "`Node1`" and "`Node2`" are branch nodes; "`Node3`" is either a branch node or a leaf node.

A trailing "`/`" is optional. "`/WebXi/`" and "`/WebXi`" are the same node.

3.3 Case insensitive

All node paths and keywords are case insensitive. It is up to the specific device whether data and arguments are case insensitive or not.

3.4 JSON

The body part of REST commands and of replies will be transmitted as JSON (see http://www.json.org). The encoding is ***Utf-8***.

The body for a leaf node contains value for this node.
The body for a branch node is an object containing values for all immediate children.
Child leaf nodes are returned fully. Child branch nodes will only be returned if the "*recursive*" keyword is used. If not, then the returned information will only contain the names of the branch nodes, their values will be **null**.

On **PUT** methods the body may (but is not required to) contain information about children. The information will be set if given.
Resources may be left out when setting values; only resources specifically mentioned will be modified.

3.4.1 Examples

Consider the following REST structure:

/WebXi		(Branch node)
/WebXi/a		(Branch node)
/WebXi/a/b	(value 2)	(Leaf node)
/WebXi/a/c		(Branch node)
/WebXi/a/c/d	(value 4)	(Leaf node)

`GET /WebXi/a` returns:

```
{
      "b": 2,
      "c": null
}
```

`GET /WebXi/a?Recursive` returns:

```
{
      "b":  2,
      "c": {"d":  4}
}
```

`GET /WebXi/a/b` returns:

```
2
```

`GET /WebXi/a/b?Recursive` returns:

```
2
```

`GET /WebXi/a/c` returns:

```
{"d": 4}
```

`GET /WebXi/a/c?Recursive` returns:

```
{"d": 4}
```

GET /WebXi/a/c/d returns:

```
4
```

`GET /WebXi/a/c/d?Recursive` returns:

```
4
```

`PUT /WebXi/a/b` that sets the b's value:

```
22
```

`PUT /WebXi/a` that also sets the b's value:

```
{"b": 22}
```

`PUT /WebXi/a` that changes both b and d:

```
{
      "b":  22,
      "c": {"d": 44}
}
```

`PUT /WebXi/a` that only changes d:

```
{
      "c": {"d": 44}
}
```

3.5 Keywords

A request may contain a query string. This query string contains one or more keywords or keyword/value pairs.
These are specified in the normal HTTP way[1]. The first keyword is preceded by a question mark ("?"), any following keywords are preceded by an ampersand ("&").

A keyword may have a value, they are specified by using an equal sign ("=") followed by the value, e.g.

PUT /WebXi/Applications/FFT?Action=Start

The spelling of all keywords is case insensitive.

This table lists the available keywords and whether the keywords must be implemented by a device or whether they are optional:

Keyword	Description	Comment
`Recursive`	Request value a node and its of children	
`Indent`	Request that child nodes are indented	
`Action`	Specifies an action to perform	
`Argument`	Specifies an optional argument for an action	Can only be specified together with `Action`
`Sync`	Requests that a sync event is sent when the request has completed.	
`Password`	The device's password	Required if the device is password protected

3.5.1 Recursive

It is possible to request a recursive data structure by specifying the "`Recursive`" keyword on the URI:

```
GET /WebXi/a/b?Recursive
```

Certain node types may be left out of a recursive GET, the client will have to get those explicitly. Such nodes marked with the RecursionExcluded flag in the metadata (see also 5.2.7).

3.5.2 Indentation

The JSON returned by GET requests is not meant to be read by humans and is not indented for easy reading. If you want to see the result of a request indented, then add the "`Indent`" keyword.

Here are some examples:

```
GET /WebXi?Indent
GET /WebXi?Recursive&Indent
```

3.5.3 Action and Argument

Actions, such as starting a measurement or starting the generator are done using the "`Action`" keyword:

```
PUT <URI>?Action=<ActionName>&Argument=<Argument>
```

[1] See the section about Query at Wikipedia: Uniform Resource Locator

Where <*URI*> is the address of the relevant node, and <*ActionName*> describes what to do.

It is possible to specify an optional argument using "`&Argument=`<*argument*>". The Argument keyword can be left out if there isn't any argument.

Action names are not case sensitive. It depends on the device whether the arguments to a given action is case sensitive.

Example:
To pause or continue a measurement on a Type 2245 Sound Level Meter:

```
PUT <URI>/WebXi/Applications/SLM?Action=PauseContinue
```

3.5.4 Sync

The requester can use this to synchronize streams and commands:
If an application receives a sync event with a given sync id on a stream, then it can be sure that all events pertaining the command with the sync id has been received.

The server will send a Sync event just before completing the request if this keyword is specified. The format is:

`&Sync=`<*SyncId*>

Where SyncId is a positive value (0 is reserved).

3.5.5 Password

This keyword is required in the request if the device is password protected.

The format is:

`&Password=`<*password*>

If the specified password does not match the device's password, then the response will be an error message saying that the request isn't authorized.

3.6 Versioning

The client can request that a specific version of the protocol is used. This is however only a request, and the device may or may not take this into consideration when choosing which version to use.

In general, all changes to the protocol should be backwards compatible (e.g. by only adding optional resources to the protocol).
If a more complex change is needed, then the version of the protocol will be changed.

Versioning is handled using content negotiation:

- The client sends a request that includes the *X-WebXi-Version* header. (This is optional)
- This tells the device which version of the protocol the client wishes to use. The device then chooses a protocol version (which may be different from the client's request).
- The device returns a response with the *X-WebXi-Version* header set to the protocol version selected by the device.
- The client may then inspect the response header to determine the version used.

Note that the X-WebXi-Version header is optional in the request; this is to allow for a normal web browser to query the device**. All applications should, however, specify the version in the request to be more stable for version changes.**

Example:

```
===>
GET /WebXi/Inputs/1/SampleRate HTTP/1.1
X-WebXi-Version: 1.0
<===
HTTP/1.1 200 OK
Content-Type: application/json
X-WebXi-Version: 1.0

...response data in version 1.0 format here...
```

Currently there is only one version of the protocol: `1.0`

3.7 Caching

Caching of responses should be turned off by including this header in all requests:

```
Cache-Control: no-cache
```

4 WebXi Tree structure

All WebXi devices should adhere to the same general layout of nodes as much as possible. This chapter gives guidelines for this tree structure.

The basic layout looks like this:

- **/WebXi**
  - **[Applications]**
    - **SLM (*)**
      - ***State***
      - [Application settings]
  - **[Channels]**
    - **Analog (*)**
      - ***State***
      - **[Inputs]**
        - **1 (*)**
          - [Channel settings]
      - **[Outputs]**
        - **1 (*)**
          - [Channel settings]
  - **[Sequences]**
    - **SLM (*)**
      - **1 (*)**
        - Descriptors
    - **FFT (*)**
      - **2 (*)**
        - Descriptors
  - **Device**
    - ***Access***
      - ***[Password]***
      - ***[Challenge]***
    - ***Class***
    - ***Family***
    - ***Description***
    - ***[SerialNumber]***
    - **[Type]**
    - **Version**
      - **Firmware**
      - **[Hardware]**
      - **[System]**
    - ***TimeFamily***
    - ***StartTime***
    - ***[Mode]***
    - ***[Time]***
    - ***[Issues]***
    - [Device settings]
  - **[Streams]**
    - **1 – My Stream (*)**
      - *Stream settings*

Nodes marked with (*) are example nodes. Nodes in brackets are optional.
The following sections go into more detail on the use of these nodes.

4.1 /WebXi

This is the top node of the WebXi node tree. It is mandatory.

4.2 /WebXi/Applications

This optional node contains nodes for any application that the device may support.
An example is "SLM" which would contain all settings and nodes specific for the SLM application.

4.2.1 Application State and State Actions

This node contains the current state of the application, e.g.:

```
/WebXi/Applications/SLM/State
```

The state node is a read only text node that contains the current state of the application.

Actions that change an applications state should be implemented on the application node, e.g.:

```
PUT /WebXi/Applications/SLM?Action=PauseContinue
```

The possible states and actions leading to these states depends on the device, so see the device specific documentation for this.
However, these are the most basic states:

- **Deactivated:** The application is not running, and all settings can be changed.
- **Activated:** The application is ready to run, settings have been applied and /WebXi/Sequences (see below) is updated. Settings may be changeable, see device documentation for details.
- **Running:** The application is running/measuring.

4.3 /WebXi/Channels

This optional node contains nodes for all input and output channels that the device supports.
The channels may be divided into groups such as "Analog" or "CAN" and further subdivided into Inputs (i.e. from transducer to device) or outputs (from device to transducer).
Each channel is denoted by a number and this node is the parent of all settings for the channel.

4.3.1 Channel State and State Actions

If the channels support one or more of the **Start**, **Stop**, **Deactivate** or **Activate** actions, then these actions should be implemented on the channel group, e.g. on `/WebXi/Channels/Analog`.
There should also be a **State** node under the group, e.g. `/WebXi/Channels/Analog/`State, that contains the current state.

4.3.2 TEDS detect action

If a group of channels supports TEDS transducer detection, then there should be a **Detect** action on that group, e.g. on /WebXi/Channels/Analog/Inputs.
It is done on this level and not on the Analog level, because we do not typically want to also do a TEDS detection on `/WebXi/Channels/Analog/Outputs`.

4.4 /WebXi/Sequences

This section of the node tree contains information about all the sequences that can be streamed (See chapter 9 for more details on streaming).
The tree is organized with a branch for each application or channel type. The sequences are organized under this node and have a number as name. This number is used when setting up a stream.
The sequences may be organized into one or more levels of sub nodes; for instance, SLM organizes sequences into types, e.g.:

```
/WebXi/Sequences/SLM/Logging/6
/WebXi/Sequences/SLM/Total/501
/WebXi/Sequences/SLM/Instantaneous/60
```
Etc.

Each sequence node has *descriptor* nodes below it that describe the sequence.

Example:

- `/WebXi/Sequences`: All sequences are organized under this node.
- `/WebXi/Sequences/SLM`: Sequences from the Sound Level Meter
- `/WebXi/Sequences/SLM/Logging`: Logging sequences from the Sound Level Meter
- `/WebXi/Sequences/SLM/Logging/6`: Specific logging sequence
- `/WebXi/Sequences/SLM/Logging/6/Name`: The name of the sequence
- `/WebXi/Sequences/SLM/Logging/6/PeriodTime`: The period time of this sequence.
- Etc. etc.

The possible descriptors are described in Chapter 11.

4.5 /WebXi/Streams

This section of the node tree contains information about created streams. A stream is a runtime connection to a device that is used to stream data from one or more sequences to or from a client. See **Error! Reference source not found.** for more details.

4.6 /WebXi/Device

This section contains information about the device, such as type numbers, serial numbers etc.

The nodes listed below are required, but only if there's relevant information to put into them.
If the device has settings etc. that concerns the device as a whole and which is not part of an application, such as for instance battery level, then it should be placed in this section.
Here is a truncated example from a Sound Level Meter:

- **Class: "**Analyzer"
- **Family: "**SLM"
- **Description:** "Sound Level Meter"
- **SerialNumber:** "1234"
- **Type: "-2245---"**
- **Version:**
  - **Firmware: "1.0.0.0"**
  - **Hardware: "0.3.0.0",**
  - **System: "0.3.0.596"**
- **TimeFamily:** 536870912
- **StartTime:** 6373872058723467264
- **Time: "**2019-01-10T08:29:48Z"
- **Issues:**
  - **Current:**
    - **Description:** "No error recorded"

Name: WebXi for SLM version 1.0
Author: Helge Rasmussen
Protocol version: 1.0
Document version: 1.0

        - **Error:** "None"
    - **Last:**
    - **Reports:**
        - **2016-11-24T07:02:43Z:** [null]
        -

4.7 /WebXi/Device/Access

This node contains information used to access the device.
The node is the parent of one or more of the following nodes:

- **Password:** An **optional** "Write-only" node that exists if the device has password protection. If it exists, then the user can change the password by changing this node. The password will not be visible (shown as "******").
- **Challenge:** A proprietary node used by HBK devices.

4.7.1 /WebXi/Device/Class

The class of the device. Possible values are:

- **Analyzer**: A hand held analyzer.
- **Frame**: A frame
- **Module**: A module (which may or may not be in a frame)

4.7.2 /WebXi/Device/Family

The family of the device within the given class.

Possible values:

- LANXI: A LAN-XI module
- SLM: A Sound Level Meter.

4.7.3 /WebXi/Device/Description

A short, one line, text describing the device.

4.7.4 /WebXi/Device/SerialNumber

The serial number of the device.

4.7.5 /WebXi/Device/Type

This node contains the type description of the device.
It has the format "<Prefix>-<Number>-<Model>-<Variant>-"
Where:

- **Prefix:** Prefix for the type number (can be empty)

4.7.6 /WebXi/Device/Version

This node contains version information, in the format "<Major>.<Minor>.<Patch>.<Build>".
There can be sub nodes for **System**, **Hardware**, **Firmware**.

4.7.7 /WebXi/Device/Licenses

This optional section contains information about installed licenses.

Each license is represented by a sub node, and each license contains two nodes:

- **Description:** A description of the license
- **Key:** The license key

4.7.8 /WebXi/Device/TimeFamily

The time family used when specifying times in binary 64-bit format. The time family specifies the size of a clock tick, see Chapter 8 for details.

4.7.9 /WebXi/Device/StartTime

The time at device boot specified as binary 64-bit time (see Chapter 8). All streamed data is guaranteed to have timestamps after this time.

4.7.10 /WebXi/Device/Time

The current time in the device expressed in text format, e.g. "`2017-01-12T11:29:52Z`".

This time is in UTC (Zulu) time.

4.7.11 /WebXi/Device/Issues

This optional node is used to examine issue logs on the device. Such logs are typically created after a crash, or upon request.

It contains the following sub nodes:

- **Current:** Node that contains information of a problem that currently exists on the device:
  - **Description:** Text description of the error
  - **Error:** Error type
- **Last:** Information about the last issue on the device:
  - **Description:** Description of the issue
  - **Error:** Error type
  - **CorrectiveAction:** What happened after the issue was detected (e.g. PoweredDown)
  - **ReportGenerated:** True if an error report was generated under reports
- **Reports:** A log of error reports created on the device. Each error report consists of a node, the value is of no importance (always empty array: "[null]"). The nodes are named with the UTC time of the issue (e.g. "2016-11-24T07:02:43Z"). Issuing a HTTP GET on the node will return a text log concerning the issue.

4.8 Guidelines for representing object model in WebXi

4.8.1 Selection of a branch in the object model

If only one branch can be active at the same time:

When a node has several child nodes, and only one of these can be active at a time, then the active node is chosen by a leaf node with the name "`Select`". Its value is the name of the active node.

Example:

Name: WebXi for SLM version 1.0
Author: Helge Rasmussen
Protocol version: 1.0
Document version: 1.0

.../Functions/Select	(value "Preamp")
.../Functions/Direct	(has child nodes)
.../Functions/Preamp	(has child nodes)
.../Functions/CCLD	(has child nodes)

In this example the active function is "Preamp", the other possibilities ("Direct" and "CCLD") are not active in the hardware. The user may set values on these nodes, but these values will not be used until the relevant node is selected.

If more than one branch can be active at the same time:

When a node has several child nodes, and more than one can be active at a time, then the child nodes all have a leaf node with the name "Active". Its value is true for the active nodes, false for others.

Example:

.../Analog1/Input1	(has child nodes)
.../Analog1/Input1/Active	(value "true")
.../Analog1/Input2	(has child nodes)
.../Analog1/Input2/Active	(value "false")
.../Analog1/Input3	(has child nodes)
.../Analog1/Input3/Active	(value "true")

"Input1" and "Input3" are active in this example.

4.8.2 PUT errors and constraint handling

If there are constraints between nodes (such as the sum of the values must be less than something), then an error (409 "Conflict") should be issued when the constraint is broken. The device should not try to modify values to satisfy the constraint.

If a PUT fails, then it will return one of the following errors:

- **403 ("Forbidden"):** The node cannot bet set at this time (because of e.g. wrong state).
- **409 ("Conflict"):** A constraint between several values would be broken if the node is set.
- **400 ("Bad Request"):** The node cannot be set to the given value.

If one of these error codes is returned, then the answer may contain one or more of the following information:

- Whether the request was partially performed, or it was not performed at all.
- The address of the resource where the update failed.
- A message in English describing what went wrong.

This is a JSON object:

```
{
      "Partial": <True or false>,
      "URI": <Uri>,
      "Error": <Error message>
}
```

The "Partial" field is set to true if the request is a compound update (more than one node is set), and the update was only partially performed. It will be false if none of the nodes were changed. All fields are optional and can be left out.

The above response may be left out if no extra information is available; typically, a "Forbidden" request may be returned without a body.

5 Metadata

A WebXi device can deliver a description of the data in the object model (aka the REST structure). This description can contain the following:

- A description of the node in English.
- The data type of the node.
- Allowable values for the node (e.g. an interval or a list of legal values)
- Whether the node is read only
- What actions are supported on the node
- A list of child nodes.
- A localized name of the node

5.1 How to get metadata

It is possible to request metadata using the GET command with the **Metadata** keyword, e.g.

```
GET /WebXi/Login?Metadata&Recursive
```

The result of the above request will be a metadata description of the Login node (and all its children since the request was recursive). These metadata may vary over time.

In general, the format of the metadata command is

```
GET <URL>?Metadata[=<MetadataTypes>][&Recursive][&Indent]
```

Here <MetadataTypes> is a comma separated list of what type of metadata to get. i.e. one or more of:

- `Description`
- `Actions`
- `DataType`
- `Flags`
- `LocalName`
- `Value`

If `Value` is specified, then the values of nodes will be returned just as if it was metadata. This is a way to request metadata and values in one request.

Leaving out the metadata types means that all meta data except for Values will be returned. This is the same as specifying metadata type `All`.

5.2 Metadata structure

The metadata is structured exactly as normal data except that the value of a node is an object containing a Metadata object with the actual metadata.

Here is a simple example:

```
{
	"Name": "Donald",
	"Address":
	{
		"Street": "1313 Webfoot Street",
		"City": "Duckburg"
	}
}
```

Corresponding metadata structure (Only listing "Description"metadata):

```
{
	"Name":
	{
		"Metadata":
		{
			"Description": "Name of the person"
		}
	},
	"Address":
	{
		"Street":
		{
			"Metadata":
			{
				"Description": "Street name"
			}
		},
		"Town":
		{
			"Metadata":
			{
				"Description": "Town name"
			}
		}
		"Metadata":
		{
			"Description": "Address of the person"
		}
	}
}
```

5.2.1 Metadata Syntax Notation

The following sections describe the syntax of metadata using Wirth syntax notation (see https://en.wikipedia.org/wiki/Wirth_syntax_notation).

The quotation marks around JSON names have been left out to not confuse them with the quotation marks from the syntax notation.

5.2.2 Metadata Nodes

This section describes the metadata node:

Metadata*Node* = "**Metadata: {"** Metadata*Entry* { "**,**" Metadata*Entry* } "**}**"

MetadataEntry = *LocalName* |
Description |
DataType |
IsVector |
Flags |
Actions |
Domain |
Value

These possibilities are described in the following sections.

5.2.3 LocalName

This optionally gives a localized name of the node:

LocalName = "**LocalName :**" *LocalNodeNameString*

LocalNodeNameString is a localized name of the node.

It is only available if the device supports localization.

5.2.4 Description

This gives the description of the node:

Description = "**Description :**" *DescriptionString*

DescriptonString is the actual description.

5.2.5 DataType

This specifies the data type of the node. If the node is a vector, then this is the data type of each element in the vector.

Possible values are **Float** (a 32 bit float), **Double** (a 64 bit float), **Int8**, **Uint8**, **Int16**, **Uint16**, **Int32**, **Uint32**, **Int64**, **Uint64**, **Complex32** (32 bit float complex), **Complex64** (64 bit float complex), **String** and **Boolean**.

DataType = "**DataType :**" *DataTypeString*

5.2.6 IsVector

This is set to **true** for vectors. If it isn't specified or specified as **false**, then the node contains a scalar.

IsVector = "**IsVector :**" *"true"* | *"false"*

5.2.7 Flags

This optionally specifies flags that describe special behaviour of the node:

Flags = "**[**" *FlagString* { "**,**" *FlagString* } "***]***"

Possible flags are:

- **ReadOnly:** The node is read only and cannot be modified.
- **WriteOnly:** The node is write only, i.e. its value is invisible, and will always be returned as "null". If the node is a directory, then all children are invisible.
- **AlwaysVisible:** The node is always visible, even when the request has not been authenticated. If a node always is visible, then its parents are as well.
- **RecursionExcluded**: The node's value will not be returned in a recursive GET. A Branch node will show up as empty: "{}"
- **Delete:** Delete may be done on this node.
- **Post:** Post may be done on this node.
- **Directory:** This node contains a directory of files and directories. Directory nodes have an action, "**GetArchive**", which can be used to fetch the contents of the directory and its children. See Chapter 6 for more information about this.
- **File:** This node is a file. GET directly on the node will return the contents of the file. A recursive get on one of its parent nodes will return information about the size of the file. See Chapter 6 for more information about this.
- **ChangesOtherNodes:** Changing this node's value will result in new values for other nodes as well (i.e. it is probably a good idea to refresh other node values from the device).
- **EditWhileActivated:** The value of this node may be modified while the application state is activated or "higher" (i.e. NOT deactivated)
- **ReportChange:** Changes of this node should result in a Node Event (See section 9.5.7)

5.2.8 Actions

This optional section describes the possible actions that can be performed on the node:

Actions	= "**Actions : [**" *Action* { "**,**" *Action* } "***]***"
Action	= "**{**" *ActionName* "***,***" *ActionDescription* { "**,**" *ActionArgument* } "***}***"
ActionName	= "**Name :**" *ActionNameString*
ActionDescription	= "**Description :**" *ActionDescriptionString*
ActionArgument	= "**Argument :**" *ActionArgumentName*

The *ActionName* is the name that is used in WebXi when the action should be performed.

ActionDescription is a text describing the action.

If the action has arguments, then these are listed. The *ActionArgumentName* is the name of the argument.

5.2.9 Licenses

The licenses section describes what licences allow write access to this node and it children.

Licenses	= "**Licenses : [**" *LicenseString* { "**,**" *LicenseString*} "***]***"

The license defines whether the user may change the node, execute actions on a node or any of its child nodes; in short use any functionality presented by the given node and its children.

If more than one license is listed, then any of the licenses is enough to get access.

As already mentioned, licenses on a node is inherited by all its children. Thus, you will need a license for all parent nodes and for the actual node to have write access to a node and its functionality.

If no licenses are listed, then there's no special license requirement for that node (except for the requirements defined by parent nodes).

5.2.10 Domain

The domain section describes the value domain of a node. There are two types of domains **List**, which defines a list of valid values and **Interval** which identifies a range of valid values.

Domain = "**Domain : {**" *DomainElement* { "**,**" *DomainElement* } "***}***"

DomainElement = "***[***" *Interval* **"]"** | "**{**" *List*{ "**,**" *List* } "***}***"

5.2.11 List

A **List** describes a list of valid values:

List = "**List : {**" [*Names* "**,**"] *Values* "***}***"

Names = "**Names: {**" *ElementName* { "**,**" *ElementName* } "**}**"

Values = "**Values: {**" *ElementValue* { "**,**" *ElementValue* } "**}**"

A list consists of one or more List Elements. Each element defines a valid value and a name for this value. The name can for instance be used to display a list to a user.

5.2.12 Interval

This describes an interval of valid values:

Interval = "**Interval: {**" *IntervalEntry* { "**,**" *IntervalEntry* } "***}***"

IntervalEntry = *Low* | *High* | *StepSize* | *Type*

Low = "**Low:**" *LowValue*

High = "**High:**" *HighValue*

StepSize = "**StepSize:**" *StepSize*

Type = "**Type:**" *IntervalType*

LowValue and *HighValue* defines the limits of the interval (both values are included).

StepSize defines an optional step size between values and *IntervalType* specifies whether the step size is linear or logarithmic (**"Linear"** or **"Logarithmic"**). Default for type is Linear.

Intervals can of course only be defined on nodes that contain scalar numbers

6 General Actions

Applications may define their own actions, but the WebXi protocol also specifies certain general actions that may be available on some or all nodes[2].

6.1 SetFlag (on all nodes)

The **SetFlag** action can be used to set or reset certain flags on any node in the tree.
Currently the only flags that are changeable are: **ReportChange** and **RecursionExcluded**.

Name: SetFlag
Argument: <flag>=true|false
Example: `PUT /WebXi/someNode?Action=SetFlag&Argument=ReportChange=true`

Notes: The action is case insensitive. The action is not documented in metadata.

6.2 GetArchive (on directory nodes)

This action is described in the chapter about directory nodes (see 7.2.2).

6.3 Log (on /WebXi)

The Log action can be used to request that a message is written to the system log.

Name: Log
Argument: <text>
Example: `PUT /WebXi/someNode?Action=Log&Argument=SomeText`

Notes: The action is case insensitive.

[2] Not all actions are listed here. Those that are for HBK internal use are listed in the Addendum document.

7 File and Directory nodes

Some nodes in the tree may have file or directory characteristics and are marked with the **File** or **Directory** metadata flags. This chapter describes the special functionality that is available on these nodes.

7.1 File nodes

File nodes are marked with the **File** metadata flag. A file node contains data just as a file contains data, these data can be fetched by executing a GET request directly on the node:

GET /WebXi/Examples/AFileNode

Returns the content of the node. This content is probably not in json format and may even be binary.

Using the "*?Metadata=Value*" keyword will not return the content of the file since this command is expected to return valid Json. In this case, the value will be returned as "{}"; this can be used to check for the existence of a file without having to fetch its content.

GET /WebXi/Examples/AFileNode?Metadata=Value

Returns "{}" if the file exists.

The content of a file node will not be returned on a GET on one of the file node's parent nodes; in this case the node value will contain the size in bytes of the content:

GET /WebXi/Examples

Could return something like this:
{ "AFileNode" : "File: 1234" }

Here 1234 is the number of bytes in the file.

7.2 Directory nodes

Directory nodes are marked with the **Directory** metadata flag. A directory node contains other File or Directory nodes; a GET on a directory node will show child nodes just as a get on any other branch node.
Directory nodes allow for three special operations:

- It may optionally be possible to create a file in the directory by using POST (see 7.2.1).
- It is possible to get the content of a file in the directory using GET (just as on file nodes).
- It is possible to get the full directory tree by using the **GetArchive** action (see 7.2.2).

7.2.1 Creating a file using POST

It may be possible to create a file in the directory using a POST request:

POST /WebXi/…/SomeDirectoryNode/<Subdirectory>?Name=<Filename>

The body of the POST will become the content of the file.

Here <Subdirectory> is an optional path to a subdirectory under the directory node and <Filename> is the desireds name of the file. If <Filename> is left out, then the name "*postedfile*" is used.

If the file already exists, then a number will be appended to the filename to make the name unique.

7.2.2 Getting directory content using GetArchive

Directory nodes allow for a **GetArchive** action. The result of performing this action is the content of the directory and its children as a binary archive:

```
PUT /WebXi/Examples/ADirectoryNode?Action=GetArchive
```

Results in a reply containing an archive of the directory's content.

Archive format

This section describes the data that is returned as the result of a GetArchive action.

Header:
The file always starts with a header:

Contents	Type	Occurrences	Comment
`0x72E176FB`	Int32	1	Magic number that identifies this as an archive
Version	Int32	1	Archive format version: Currently always 0
Number of files	Int32	1	Number of files in the archive
File path length	Int16	Once per file	Length of the file path, in number of characters.
File path	String	Once per file	The relative path to the file from the node where the GetArchive action was performed. Slash (/) is used between parts of the path
Offset	Int64	Once per file	Offset into the reply body (in bytes) where the file content start, relative to the start of the archive reply.
Length	Int64	Once per file	Length of the file's content (in bytes)

File Data:
The reply's body then contains the file data at the Offset specified in the header. The "file" ends after the given Length. The next file must not start before the previous file has ended. It is, however, ok to have a gap between files. The data values in such a gap is be left uninitialized.

8 Time

Time in WebXi is based on the concepts from the HBK's BKC time format, but modified to support absolute time

8.1.1 The BK Connect Relative time

The relative time in BK Connect is contained in a 64-bit integer which specifies a "tick" count. There are $2^{22} * 3^2 * 5^3 * 7^2$ ticks per second, so each tick is approximately 4.33 picoseconds. The maximum duration of this time is approximately 461 days ($2^{64} * 4.33$ picoseconds**)**

We can easily specify the following sample frequency families with this:

Name	Frequency	Frequency Range
2^n Hz family	$2^n * 3^0 * 5^0 * 7^0$	From 1 to 2^{22} Hz (4 MHz)
51,2 kHz family	$2^n * 3^0 * 5^2 * 7^0$	From 25 to $2^{22} * 25$ Hz (105 MHz)
256 kHz family	$2^n * 3^0 * 5^3 * 7^0$	From 125 to $2^{22} * 125$ Hz (524 MHz)
48 kHz family	$2^n * 3^1 * 5^3 * 7^0$	From 375 to $2^{22} * 375$ Hz (1.6 GHz)
44,1 kHz family	$2^n * 3^2 * 5^2 * 7^2$	From 11025 to $2^{22} * 11025$ Hz (46 GHz)

Here a family is defined as a group of frequencies which are equal except for a factor of 2^n.
The frequency listed in the family "name" is just a typical frequency within the family.

Note that this time wraps in 64 bit after 461 days. We are not satisfied with such a short wrap time, and we need an absolute time, so this leads to the format described in the next section.

8.1.2 The WebXi absolute time

The time for WebXi is based on the ideas in the BK Connect time described above.

The WebXi time consists of two parts:

The first part is the Family; it is a little endian 32-bit quantity defining the exponents used for 2, 3, 5 and 7 (one byte for each). This defines the size of a clock tick:

Name	Description	Size in bytes
K	Exponent for 2	1
L	Exponent for 3	1
M	Exponent for 5	1
N	Exponent for 7	1

The value of the little endian 32-bit family is:

```
(k << 24) | (l << 16) | (m << 8) | n
```

This family corresponds to a tick size of:

$2^{-k} * 3^{-l} * 5^{-m} * 7^{-n}$ **seconds**

The second part is the Count; it contains the number of ticks in a 64-bit number.
If it should be interpreted as an absolute time, then it is the tick count since 00:00:00 on 1 January 1970 (Modified Julian Day 40 587.0).

Name: WebXi for SLM version 1.0
Author: Helge Rasmussen
Protocol version: 1.0
Document version: 1.0

8.1.3 Important notes about the time format

Compatibility with BK Connect

Note that BK Connect signal analysis only supports part of the values that this time format supports.

BK Connect sample frequencies that can be expressed by:

$$2^k * 3^l * 5^m * 7^n \text{ Hz}$$

Where

$$k \leq 22, l \leq 2, m \leq 3, n \leq 2$$

Wrap

To avoid wraps the family should be chosen so that the time to wrap, which is:

$$\frac{2^{64}}{2^k * 3^l * 5^m * 7^n} \text{ seconds}$$

is longer than 100 years.

The following table contains suggested values that allow for this:

Name	Family (k, l, m, n)	Max Frequency	Time until wrap
65 kHz family	32, 0, 0, 0	4.2 GHz	136 years
51,2 kHz family	27, 0, 2, 0	3.3 GHz	174 years
256 kHz family	25, 0, 3, 0	4.2 GHz	139 years
48 kHz family	23, 1, 3, 0	3.1 GHz	186 years
44,1 kHz family	18, 2, 2, 2	2.9 GHz	202 years

A note on PTP and LXI Compatibility

The PTP timestamp format (as defined by IEEE 1588) is different from the above proposed format:

Name	Data type	Description
Seconds	6 bytes	Time in seconds. In LXI this value is split in two parts: The 4 byte "*Seconds*" field and the 2 byte "*Epoch*" field in the header
Nanoseconds	6 bytes	In LXI this is divided in the 32 bit "*Nanoseconds"* field and the 16 bit "*Fractional Nanoseconds*" field.

Zero time for this is 00:00:00 on 1 January 1970, which is the same as for WebXi format above.

Applications will have to convert to and from WebXi format when implementing LXI and PTP specific protocols.

9 Streaming

The device can deliver results, quality data and events over a TCP/IP connection or WebSocket[3] connection. It might also be possible to request that the data be written to storage on the device (such as an SD card). Finally, there is also a possibility to stream data **to** a device, for instance for generator output.

The protocol allows for several clients to request input streaming of the same or different data. Output streaming, on the other hand, is point to point only, in the sense that there is exactly one client that produces data, and exactly one application on the device that receives the data.

9.1 Managing streaming

The client configures the data streaming by using REST.
This is done by executing a POST on "/WebXi/Streams":

```
===>
POST /WebXi/Streams HTTP/1.1

{
    "ConnectionType": <ConnectionType>,
    "Name": <Name>,
    "Direction": <Direction>,
    "Sequences":
     [
        <SequenceId>, <SequenceId>, …
    ]
    "MessageTypes":
     [
        <MessageTypeName>, <MessageTypeName>, …
    ]
}
<===
HTTP/1.1 200 OK

{"URI": ["<URI new Stream Resource>", …]}
```

<ConnectionType> specifies which transport mechanism to use. SLM currently implements "**Socket**" for a TCP/IP connection and "**WebSocket**" for streaming to a Web Socket.

<Name> is a user-specified name of the stream.

<Direction> is optional and indicates whether the stream streams to or from the device. Possible values are "**ToDevice**", "**FromDevice**". The default is "**FromDevice**".

<SequenceId> identifies which sequences to stream. The id's here match the ids specified in the "**Sequences**" part of the node tree. A stream can only contain sequences with the same **StreamType** descriptor. See more about this descriptor in chapter 11.

<MessageTypeName> lists what message types to stream, the names are in quotes. Section 9.5.1 lists the possible message types and their names.

A created stream will show up under /WebXi/Streams, e.g.:

/WebXi/Streams/1/Name	Name of the stream
/WebXi/Streams/1/Direction	Direction of the stream (default: "FromDevice")
/WebXi/Streams/1/State	State of the stream ("Ready", "Open")

[3] WebSocket is a standard internet protocol defined in RFC 6455

/WebXi/Streams/1/ConnectionType	The type of the connection (“Socket”, “WebSocket”)
/WebXi/Streams/1/Port	The port that the client should open to receive data (only for “Socket” connections)
/WebXi/Streams/1/Sequences	Get/Set the sequences that should be transmitted in this stream.
/WebXi/Streams/1/MessageTypes	Get/set the message types that should be transmitted in this stream.

The nodes are read-only except for the `Sequences` and `MessageTypes` nodes.

9.1.1 Closing a stream

A stream can be closed and removed by deleting its top node, e.g.:

```
DELETE /WebXi/Streams/1
```

It is also removed from `/WebXi/Streams` if the connection (socket or web socket is closed).

9.2 Streaming to and from a device

A stream is unidirectional, and it is not allowed to mix input sequences and output sequences in the same stream. The direction of the stream is defined by the Direction parameter in the POST that creates the stream.
An output sequence (i.e. streaming **to** a device) is recognized by the optional “**Direction**” descriptor, which is “**ToDevice**” for output sequences. The descriptor’s default is “**FromDevice**”.

9.3 Timing of messages in an input stream

Each message in the stream contains a time stamp, this is however not enough information for the client to efficiently handle the data.

For instance, the client will need to

- Match quality data to sample data
- Be sure to not discard any data before knowing if a trigger event is happening in the data.

The device must follow these guidelines to help the application with these tasks:

- Time for any given sequence must never decrease.
- Data describing other data must be transmitted before the data they describe (e.g. quality of sequence 1 before sequence 1.)
- General status data must be sent before any of the input data.

9.4 Data types and value domains

Data types below are often specified as being signed, even when the value domain obviously is in fact unsigned (such as the number of values in a message). The reason for this is to make it possible to be compliant with the Microsoft “Common Language Specification” (CLS).

It will of course be an error to specify a negative number of values for a count even though it in fact is possible.

Ids (such as **SequenceId**) are also signed, in this case all values except for 0 are valid ids unless otherwise specified.

The value 0 is used to indicate an unknown Id.

9.4.1 Strings

Strings are in UTF-8 format. They are represented as a byte count followed by that number of bytes in a stream:

Name	Length (in bytes)	Contents
Count	4	The number of bytes (not characters) in the UTF8 string as an Int32. If Count is 0, then the string is empty. Count may not be negative.
Bytes	Count	The actual content of the string. (Note that Count may not equal the number of characters in the string since a single character may be from 1 to 4 bytes in length).

9.4.2 Booleans

Booleans are in REST/JSON represented as true/false. They are represented by a 16-bit value in streaming. 0 means false, non-zero means true

9.5 Streaming message format

The stream consists of *messages* transmitted after each other. The format of a message is a header optionally followed by some content:

Name	Length (in bytes)	Offset	Contents
Magic	2	0	`0x4b42` as little endian 16-bit quantity
HeaderLength	2	2	The length of the rest of the header up to but not including "ContentLength". Currently this is 16[4].
MessageType	2	4	Identifies the content of the message
ContentVersion	2	6	The version of the content field. Since Content is different for each message type, the content version also may vary from message type to message type.
Reserved2	4	8	Reserved. Set to 0
Time	8	12	Time of message
ContentLength	4	20	Length of the message content in bytes

This header is followed by the content part of the message:

Content	ContentLength	24	Depends on the message type.

The actual content depends on the message type. See the chapter on Message Types below.

All multi-byte values in a message are transmitted in little endian byte order.

[4] If new fields are needed in the header, then they should be appended after the "Time" field. Existing fields must never be removed from the header, and the length of the header should always be a multiple of 4 bytes. If these rules are followed, then the header length will always increase for each new header version. The client can use HeaderLength as a kind of Header version field.

9.5.1 Message Types

This section describes the different types of messages that can be sent to the client.

Message Type	Value	Type	Description
SequenceData	1	S	Data values from a sequence.
AuxSequenceData	11	S	Data values from a sequence optimized for non-time-equidistant data.
DataQuality	2	S	Indicates the data quality of a certain sequences data. The quality message is typically only generated when the quality of a sequence changes.
State	3	E	Indicates the state of the application.
Status	4	E	Status event, such as PTP sync lost.
Trigger	5	S	Trigger events from a sequence
Node	6	E	Sent when the value of a node with ReportChange flag has changed.
Sync	7	E	Sync event: Sent when using the Sync keyword
	8		Reserved for future use
Debug	9	E	A message containing debug information

Type "S" indicates that the message refers to a specific sequence, "E" indicates that the message is an "event" and does not refer to a specific sequence.

Each message type has an associated data structure that is sent with the message.

The following sections describe the data transmitted for the different message types.

9.5.2 SequenceData

Messages of this type contain data values from a sequence.

The format of the message depends of the value of the *MessageFormat* descriptor for the sequence as described below.

The message content is:

Name	Stream type	Description
NumberOfBlocks	Int16	Number of blocks with data in this message.[5] The number of blocks in the message must be greater than zero.
MessageFormat	Int8	Raw:0, Mp3:1, Flac:2
Reserved	Int8	Reserved. Set to 0

The header is followed by *NumberOfBlocks* blocks, the block format depends on the MessageFormat:

Block structure in the "Raw" message format (default):

Name	Stream type	Description
SequenceId	Int16	Identifies the sequence that produced the following values.
ValueLength	Int32	Number of bytes in "Values".
Values	Array of Values	Data from the sequence. In the case of a vector a value means a single vector.

[5] It is up to the device to decide whether it wants to group sequences into one message or not.

The block structure in any other message format
In this case the content is a list of the sequences included in the frame followed by the number of bytes in the frame and a frame in the specified format containing data for the sequences:

Current available formats: **MP3**, **Flac**

Name	Stream type	Description
NumberOfSequences	Int16	Number of sequences in this block.
SequenceIds	Array of Int16	*NumberOfSequences* ids that are contained in the frame below
FrameLength	Int32	Number of bytes in the frame
Frame	Array of bytes	Content of a Frame in the specified format containing data from the sequences specified above.

Note:

- A sequence may be sent with no sequence data (i.e. "ValueLength" or "FrameLength" 0). This indicates that the time of the sequence has reached the time specified in the header. This may be used by non-periodic sequences to report that there will be no data at or before the given time. (What in BK Connect signal analysis parlance is called a Synchronade).

9.5.3 AuxSequenceData

Messages of this type contain data values from a sequence.

It can contain the same types of data as the SequenceData messages, but while SequenceData messages are optimized for time equidistant data, this message type contains a relative time, and thus is better suited for non-equidistant data.

The message content is:

Name	Stream type	Description
NumberOfSequences	Int16	Number of sequences with data in this message.[6] The number of sequences in the message must be greater than zero.
Reserved	Int16	Reserved. Set to 0

The following structure is then repeated "NumberOfSequences" times:

Name	Stream type	Description
SequenceId	Int16	Identifies the sequence that produced the following values.
NumberOfValues	Int16	Number of values.

The following structure is then repeated "NumberOfValues" times:

Name	Stream type	Description
RelativeTime	Int32	Time in ticks since the absolute time specified in the header
Values	Array of values	Data from the sequence. In the case of a vector a value means a single vector. This means that the actual number of scalar values (e.g. floats) transmitted is NumberOfValues * VectorLength from the interpretation message.

Note:

- A sequence may be sent with "NumberOfValues" 0. This indicates that the time of the sequence has reached the time specified in the header. This may be used by non-periodic sequences to report that there will be no data at or before the given time. (What in BK Connect signal analysis parlance is called a Synchronade).

9.5.4 DataQuality

Messages of this type contain quality information for one or more sequences. The quality data contains a "quality id"; the quality data applies to sequences with a "QualityId" descriptor with this quality id.

[6] It is up to the device to decide whether it wants to group sequences into one message or not.

The quality message is typically only generated when the quality of a sequence changes.

The message content is:

Name	Stream type	Description
NumberOfQualities	Int16	Number of qualities in this message.
Reserved	Int16	For future use; set to 0

The following structure is repeated "NumberOfQualities" times:

Name	Stream Type	Description
QualityId	Int16	The following validity applies to sequences with this quality id.
Validity	DataValidity flags (Int16)	Quality info
Reserved	Int32	For future use; set to 0

DataValidity flags (Int16)

The DataValidity flags (Int16) is exactly as it is in BK Connect. The values are "flags" that can be OR'ed together if more than one condition exist.

Name	Value	Description
Valid	0	Data is valid
Unknown	1	Data quality is unknown
Clipped	2	Data is clipped
Settling	4	Data is settling
Invalid	8	Data is invalid for another reason than the above.
Overrun	16	Overrun happened right before the data value with the same timestamp as this DataQuality message.

9.5.5 State

This message contains the state of a application. It is sent when a stream containing state information is opened, and when the state changes.

The message content is:

Name	Stream type	Description
Application Name	String. See 9.4.1	The name of the application.
State	State enum (Int16)	16-bit enumeration of the possible states.
	Int16	Reserved for future use (was subsystem).

State enum (Int16)

Possible values for "State":

Name	Value	Description
Initial	0	Initial state when system starts
Error	1	The application ends up here in case of serious errors.
Deactivated	2	The application is deactivated
Activated	3	The application is activated and ready to be started
Running	4	The application is running
RampingDown	5	The application is ramping level down
Pause	6	The application is paused
Stopping	7	The application is stopping
ResultMeasured	8	The application has measured a result and it is now available
ResultLoaded	9	The application has loaded an old result that is now available

9.5.6 Status

Status event, such as PTP sync lost. This is sent when a stream containing status events is opened, and when a status change happens.

The message content is:

Name	Stream type	Description
ChannelType	Int16	Optionally identifies the channel that produced the event. If irrelevant, then ChannelType will be None (0)
ChannelId	Int16	Optionally identifies the channel that produced the event. If irrelevant, then ChannelId will be 0)
StatusType	StatusType enum (Int16)	16-bit enumeration of the possible statuses.
Reserved	Int16	Reserved for future use. Set to 0.
Value1	Int32	Content and data type depending on "StatusType"
Value2	Int32	Content and data type depending on "StatusType"
String	String. See 9.4.1	Optional String describing the status. The maximum length of the string is 64 bytes

StatusType enum (Int16)

Possible values for "StatusType":

<table>
<tr><th>Name</th><th>Value</th><th>Contents of the Value fields</th><th>Description</th></tr>
<tr><td>Unknown</td><td>0</td><td>Not used</td><td>Status not set (should never happen)</td></tr>
<tr><td>AnalogOverload</td><td>1</td><td>Value1: Overload enum</td><td>Indicates a change in analog overload status</td></tr>
<tr><td>DigitalOverload</td><td>2</td><td>Value1: Overload enum</td><td>Indicates a change in digital overload status</td></tr>
<tr><td>CCLDOverload</td><td>3</td><td>Value1: Overload enum</td><td>Indicates change in CCLD overload status.</td></tr>
<tr><td>CVLDOverload</td><td>4</td><td>Value1: Overload enum</td><td>Indicates change in CVLD overload status.</td></tr>
<tr><td>CommonModeOverload</td><td>5</td><td>Value1: Overload enum</td><td>Indicates a change in Common mode overload status.</td></tr>
<tr><td>InputProtection</td><td>6</td><td>Value1: Overload enum</td><td>Indicates a change in Input protection status. (If "Invalid", then Input Protection is active)</td></tr>
<tr><td>CableBreak</td><td>7</td><td>Value1: Overload enum</td><td>Indicates change in Cable Break status</td></tr>
<tr><td>Fan</td><td>8</td><td>Value1: Fan enum
Value2:
<table><tr><th>16-bits</th><th>8-bits</th><th>8-bits</th></tr><tr><td>Not used</td><td>Control mode</td><td>Speed</td></tr></table>Bit [0-7]: Fan Speed enum
Bit [8-15]: Fan Control Mode enum</td><td>Fan event</td></tr>
<tr><td>Temperature</td><td>9</td><td>Value1:
<table><tr><th>8-bits</th><th>16-bits</th><th>8-bits</th></tr><tr><td>Reason</td><td>Time in sec</td><td>Event type</td></tr></table>Bit [0-7]:
Temperature event type defined
in Temperature enum
Bit [8-23]:
Shutdown time in seconds.
-1 means infinite (INT16)
Bit[24-31]:
Shutdown reason defined in Temp Shutdown Reason enum.
Shutdown-reason must be ignored if shutdown-time is -1.
Value2:
<table><tr><th>8-bits</th><th>8-bits</th><th>16-bits</th></tr><tr><td>Control mode</td><td>Level</td><td>Temperature</td></tr></table>Bit [0-15]:
Temperature in Celsius (INT16).
-300 indicates error.
Bit [16-23]:
Temperature level defined in Temperature Level enum
Bit [24-31]:
Temp. Control Mode enum</td><td>Temperature event</td></tr>
</table>

Power	10	**Value1**: PowerSource enum in top 16 bit, PowerMode enum in low 16 bit. **Value2**: Current estimated power consumption in mW.	Power status
PowerAlarm	11	**Value1:** Estimated remaining time before power is gone in seconds	This status is sent when power is getting low
Synchronization	12	**Value1**: Precision in nanoseconds. **Value2**: SynchronizationState enum	Synchronization status
CommandCompleted	13	**Value1:** Completion status (0: Ok) **String:** The command that completed	This status is sent when an asynchronous command has completed
Reserved	14		Reserved for future use
Overrun	15	**Value1:** Overload enum	If Invalid, then there was a gap in the data from an input channel
MessageNotSent	16		Sent if one or more messages has been lost
Debug	17	**Value1:** Whatever the debugger wants **Value2:** Whatever the debugger wants	Sent by debug code
PowerLoss	18	Not used	Shutting down because of loss of power
CriticalError	19	Not used	Shutting down due to a critical software or hardware issue
TransducerDetect	21	**Value1:** Type of transducer event (TransducerDetectStatus enum)	Status of a transducer detect action
SLM	23	SLM Status enum	See below for details
ConnectionTimeout	24	Not used	The current user was logged out, and applications stopped, because of timeout.
Calibration	25		Indicates change in calibration status.
CurrentOverload	26		Indicates change in drive current status.
SlewRateLimit	27		Indicates change in slew rate limit status
PowerSupply	28		Indicates change in power supply status
DiskSpaceAlarm	29	**Value1:** Amount of free disk space in Mb	Sent when disk space is low

Overload enum

The Overload enum contains:

Name	Value	Description
Ok	0	No overload: Values are ok
Invalid	1	Overload: Values are invalid
Low	2	Overload: Values are too low
High	3	Overload: Values are too high
UnderRange	4	Values are too low (typically below the noise floor)
OverRange	5	Values are too high

Fan Status enum

The FAN status enum contains:

Name	Value	Description
Unknown	0	Unknown
ControlModeChanged	1	FAN Control-Mode changed (e.g. Auto <---> Manual)
Speed	2	FAN speed Changed
SilentOverrule	3	Silent overrule (Fires in manual mode only)

FAN speed enum (UINT8)

The FAN speed enum contains:

Name	Value	Description
Unknown	0	Unknown
Off	1	Off (When in manual mode corresponds to silent-mode)
Low	2	Lowest speed
Middle	3	Middle speed
High	4	Highest speed

FAN Control-Mode enum (UINT8)

The FAN control mode contains:

Name	Value	Description
Unknown	0	Unknown
Auto	1	Fans are controlled by the device
Manual	2	Fan speed is set by the client. (When overheat occurs it changes to Auto).

Temperature Status enum

The Temperature status enum contains:

Name	Value	Description
Unknown	0	Unknown
ControlModeChanged	1	Control mode changed (Reserved for test purpose)
LevelChanged	2	Temperature level changed
Overheat	3	Overheat occurred
Shutdown	4	Powerering down in 2 minutes
ShutdownCancelled	5	Power down cancelled (e.g. temperature decreasing)
TooManyReadError	6	Too many read errors in a row
AllThermoFailure	7	Read temperature from all thermometers failed

Temperature Shutdown reason enum

The Temperature shutdown reason enum contains:

Name	Value	Description
None	0	No shutdown
HighTemp	1	Shutdown due to high temperature.
NoSensor	2	Shutdown due to no temperature sensor found or read failure from all sensors.

Temperature level enum (UINT8)

The temperature-level enum contains:

Name	Value	Description
Unknown	0	Unknown
Off	1	Off (This level corresponds to Fan-speed Off)
Low	2	Low level temperature
Middle	3	Middle temperature
High	4	High temperature
Overheat	5	Overheat level
PowerOff	6	Power off level

PowerSource enum (Int16)

The PowerSource enum contains:

Name	Value	Description
Unknown	0	Unknown power source. Should not be used.
PoE	1	Power over Ethernet
DC	2	
AC	3	

Battery	4	
USB	5	
Backup	6	

PowerMode enum (Int16)

The PowerMode enum contains:

Name	Value	Description
Unknown	0	Unknown mode.
Normal	1	Normal operations mode
PowerSave	2	Power save mode.
Off	3	Power will be off after this message.

SynchronizationState enum (Int16)

The SynchronizationState enum contains:

Name	Value	Description
Unknown	0	Unknown synchronization state
OutOfLock	1	Device is not locked to anything
Locking	2	The device is trying to lock
Locked	3	The device has locked synchronization

TransducerDetectStatus enum (Int16)

The TransducerDetectStatus enum contains:

Name	Value	Description
Unknown	0	Unknown transducer detection state
Start	1	Transducer detection started
Done	2	Transducer detection succeeded
Fail	3	Transducer detection failed

SLM Status enum (Int16)

The SLM Status enum is in the lower 16 bits of value 1 when the StatusType is SLM.

Name	Value	Description
None	0	No status bits set
ProjectCreated	1	A project has been Created. String contains path to the project.
ProjectDeleted	2	A project has been Deleted. String contains path to the project.

9.5.7 Node

This event is generated when one or more nodes with the ReportChange flag has changed.

It may also be sent at other times depending on the implementation.

The message content is:

Name	Stream type	Description
NumberOfChanges	Int16	Everything below is repeated this number of times
Reserved	In16	Reserved for future use, set to 0.

The following is repeated NumberOfChange times:

Name	Stream type	Description
Flags	Int16	Bit flags: 1: Value has changed 2: Metadata has changed 4: Node has been created 8: Node has been deleted
Reserved	Int16	Reserved for future use, set to 0.
Path	String	Path to the changed node
JsonString	String	A JSON description of the changes to the node (value and/or metadata) and optionally its children. The format is the same as returned on GET requests.

9.5.8 Debug

This event contains a debug message.

The message content is:

Name	Stream type	Description
String	See Strings 9.4.1	Text message

9.5.9 Sync

This event contains a Synchronization event id. This is sent when a request contains the "`sync=`<syncId>" keyword.

This is used to synchronize streams with WebXi requests.

For instance, a client can send a request:

PUT /WebXi?Someaction&Sync=1234

Then, after successful completion of the PUT the client waits for a sync message with id 1234 (note that it may have arrived already).

At this point in time the user will know that all changes on the stream that pertain to the issued action have happened already.

The message content is:

Name	Stream type	Description
SyncId	Int32	The id from the REST request

10 Error code usage

This chapter describes which HTTP status codes the device should return, and what they mean.
In case of an error the body can contain a description of the error.

The format is:

```
{"Error": "<Text describing the problem>"}
```

10.1 101 ("Switching Protocols")

The host has accepted a WebSocket connection.
After this is received the WebSocket connection is established, and the connection is in the OPEN state.
The WebSocket protocol is described in RFC 6455 (See http://tools.ietf.org/html/rfc6455)

10.2 200 ("OK")

This is the default ok response. Use this unless 201 ("Created") or 202 ("Accepted") should be used.
Place the answer in JSON format in the Entity-Body.

10.3 201 ("Created")

Use this response when the user uses POST to create a new node (e.g. creating a stream). Return the URI for the child node

10.4 202 ("Accepted")

Use this answer when the user issues a request that does not completed at once.

10.5 400 ("Bad Request")

M or otherwise bad request.

10.6 401 ("Unauthorized")

The client isn't authorized to perform the action. Examples of use: The password is wrong.

10.7 403 ("Forbidden")

The state of the device forbids execution of the request. Example: Requesting a TEDS detect while measuring.

10.8 404 ("Not Found")

Node not found.

10.9 405 ("Method Not Allowed")

The request is not supported by this node.

10.10 409 ("Conflict"

There's a constraint between nodes that would be broken by this request.

10.11 500 ("Internal Server Error")

There is a serious problem on the device. There is a text in the body describing the problem.
This should be used for non-recoverable errors such as asserts.

10.12 503 (“Service Unavailable”)

The service is currently unavailable, typically because the device is busy serving the controlling client.

11 Descriptors

This chapter lists the descriptors that can be used in the description of sequences in the WebXi protocol. This is a superset of the descriptors used in the HBK BKC file format.

Many of the descriptors have been renamed though, here is a table of currently used descriptors and their equivalent BK Connect name:

WebXi Name	**BKC Name**
Name	GeneralName and NameInternal
FunctionType	FunctionType
DataType	DataType
VectorLength	DataAxisLength
Unit	InterpretationUnit
Scale	InterpretationScale
Offset	InterpretationOffset
IsPower	InterpretationIsPower
DataAxisType	DataAxisType
DataAxisUnit	DataAxisUnit
DataAxisLinearDelta	DataAxisLinearDelta
DataAxisLogarithmicFactor	DataAxisLogarithmicFactor
DataAxisInitialValue	DataAxisFixPoints[0].Value where DataAxisFixPoints[0].Index must be 0
PeriodTime	SequenceIndexMappingLinearDelta on the primary time sequence of the table containing the sequence. If 0, then the time axis is explicit. If non-zero, then the time axis is linear.
TimeWeighting	TimeWeightingTimeWeighting
TimeWeightingNoiseBandwidth	TimeWeightingNoiseBandwidth
IsPrimaryData (GeneralIsPrimaryData)	GeneralIsPrimaryData
FrfMode	FrfFrfMode
InputDofComponentName	InputDofComponentName
InputDofNodeId	InputDofNodeId
InputDofDirection	InputDofDirection
InputDofSense	InputDofSense
OutputDofComponentName	OutputDofComponentName
OutputDofNodeId	OutputDofNodeId
OutputDofDirection	OutputDofDirection
OutputDofSense	OutputDofSense

More can be added as they are needed.

The following WebXi descriptors do not exist in BKC:

Name	**Mandatory or optional?**	**Type**	**Description**
TableId	Mandatory	32-bit int	Number identifying the table the sequence is in. Sequences with same table id is in the same table.
TimeFamily	Optional, Default: Value of /WebXi/Device/TimeFamily	32-bit int	Time family for the sequence
Direction	Optional, default: "FromDevice"	String	Indicates the direction of the stream. Possible values are "ToDevice" and "FromDevice".
QualityId	Optional, Default: 0	16-bit int	Data quality messages with this id apply to this sequence.

ChannelType	Optional, Default: None	String	Type of channel (e.g. "AnalogInput")
ChannelId	Optional	16-bit int	Id of the channel within the given channel type
RefChannelType	Optional, Default: None	String	Type of reference channel if any (otherwise "None"). Example: "AnalogInput".
RefChannelId	Optional	16-bit int	Id of the reference channel within the given channel type.
ValueRate	Optional, default: unknown rate	32-bit int	Number of values per second. Does not need to be accurate. This may be used to estimate required buffer sizes.
StreamType	Optional	32-bit int	Only sequences with the same StreamType can coexist in the same stream. If StreamType is left out, then the sequence can coexist with all other sequences without a StreamType.
Description	Optional	String	Description of the sequence
MessageFormat	Optional, default: "Raw"	String	Message format when streaming ("Raw", "MP3" or "Flac")

12 Discovery

Devices or services that support the WebXi protocol should use Multicast DNS (mDNS) for discovery on the network.

The service name *_web-xi._tcp* has been registered with IANA[7], and should be used in all mDNS responses.

Two TXT-records have been defined as well:

- name: an optional, user-defined name for the device, e.g. "John's Sound Level Meter "
- version: the highest version of the WebXi protocol supported by the device

[7] https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml?search=web-xi

13 Document References

Ref. No.	Document Title	Description
	Leonard Richardson & Sam Ruby: RESTful Web Services	The book that describes REST protocol philosophy.
RFC 6455	The WebSocket Protocol.	Documentation of the WebSocket protocol. This is the protocol used for streaming data to and from a device.