USER GUIDE

LAN-XI Open API

BE 1872 – 15
English

Brüel & Kjær

LAN-XI Open API

User Guide

BE 1872 – 15
March 2021

www.bksv.com

Brüel & Kjær Sound & Vibration
Measurement A/S
DK-2850 Nærum · Denmark

For service and support, contact your nearest Brüel & Kjær Customer Care support team:

Headquarters: info@bksv.com,
+45 7741 2400
China (Beijing): +86 10 59935811
France: service.fr@bksv.com,
+33 1 69 90 71 02
Germany: bkservice.de@bksv.com,
+49 421 17 87 0
Italy: it.info@bksv.com, +39 02 5768061
Japan: info_jp@bksv.com, +81 3 6810 3500
The Americas: bkservice@bksv.com,
+1 770 209 6907
Spain: servicio.tecnico@bksv.com,
+34 91 659 08 20
UK & Ireland: ukservice@bksv.com,
+44 1223 389800

Go to www.bksv.com/contact for contact information to our other global offices.

Contents

Chapter 1

Introduction

This document describes the commands implemented in the REST protocol for LAN-XI devices, as well as the network streaming protocol used to stream signals to and from the devices and descriptions on how to set up and operate this.

This version of the document contains the commands that are related to the so-called "Open API".

To enable the Open API on your LAN-XI module, it must have LAN-XI Open API License BZ-5959-L-N01 installed or be running firmware version 2.10.0.344 or later.

1.1 Why LAN-XI?

A LAN-XI module offers great flexibility. At one end of the module, exchangeable front panels connect to a large variety of analog and digital signal connectors and supply transducers with the necessary current or voltage. At the other end of the module, an RJ45 Ethernet connector interfaces to your PC (typically) and to other LAN-XI modules, which may be distributed over a large area. The Ethernet cables not only handle data input/output but also supply power and synchronization at sample-level accuracy, which allows, for example, cross-spectra between any two channels in a system.

In-between the signal connectors and the network connector, you get high-quality signal conditioning, and conversion from analog to digital (and vice versa on LAN-XI modules with a generator).

Distributing the LAN-XI modules typically saves a lot of expensive and heavy analog cables when measuring on large objects, while effectively reducing ground-loop problems.

If you prefer to concentrate many measurement channels in one place, LAN-XI modules fit into frames that can be stacked or mounted in racks.

1.2 Why use Open API for LAN-XI?

BK Connect® and PULSE LabShop are Microsoft® Windows®-based software clients that provide an interface to LAN-XI modules with a large selection of analysis algorithms as well as graphic displays with advanced cursors. These applications are able to meet the needs of many LAN-XI hardware users.

But what about the following scenarios:

- You already own and maintain data-acquisition software, but want to use high-quality LAN-XI modules as front ends for a specific job.
- Your data-aquisition software interfaces to other specific hardware, company databases or asset management.
- Your data-aquisition software runs on a Linux® distribution like Debian® or SUSE®, or on Apple® OS X®.

Open API (application protocol interface) for LAN-XI is the key to all the above scenarios.

Setting up a LAN-XI system and an individual channel is simple. Open API supports a variety of setups; from single-module systems to multi-module systems to distributed systems – all sample-synchronous. It is also possible to connect modules to a wireless router and then control them from a phone or similar, but we do recommend starting with a wired solution, which is easier to debug.

1.2.1 Supported Commands, Modules and Front Panels

LAN-XI Open API will work with all LAN-XI modules with the exception of 4-ch. Input/HS-Tacho + 8-ch. Aux. Module Type 3056. Some commands are only supported for specific modules, for example, commands for bridge settings are only relevant for Bridge Input Module Type 3057.

Table 1.1 *LAN-XI module support of Open API*

Module		Open API Support
LAN-XI	Type 3050	Fully supported
	Type 3052	Fully supported
	Type 3053	Fully supported
	Type 3056	Auxiliary channels **not** supported
	Type 3057	Fully supported, including bridge configuration
	Type 3058	Fully supported, including CAN bus channels
	Type 3160	Fully supported, including output channels (generator)
	Type 3161	Fully supported, including output channels (generator)
	Type 2831	Battery modules mounted in frames are supported, see section 4.3 for more information.
LAN-XI Light*	Type 3676	Fully supported
	Type 3677	Fully supported

* LAN-XI Light modules only work as single-module systems, thus the PTP (precision time protocol) setup is irrelevant on these modules.

The numerous, detachable LAN-XI front panels will continue to work with the module to which they are attached as specified in their documentation. For example, a front panel that only works with Bridge Input Module Type 3057 on BK Connect® will also only work with this module when using the Open API.

1.2.1.1 Number of Channels

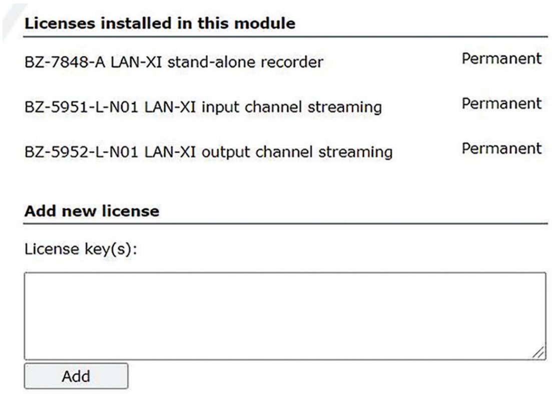
For systems with a large number of channels, LAN-XI Open API does not introduce any limiting factors into the measurement chain. If the network can handle the large amounts of data, and the application using Open API is sufficiently optimized, then systems in excess of 400 channels at 100 kHz should be easily achievable.

1.2.2 Licenses

From LAN-XI firmware version 2.10.0.344, the license check for the LAN-XI Open API License BZ-5959-L-N01 is removed. Hence any module running firmware version 2.10.0.344 (or higher) will have access to the API. When you look at the module's licenses on the module's web page, you will see the three constituents of the API as permanent licenses.

LAN-XI firmware is available from this link: bksv.com/lanxi-firmware.

Fig. 1.1 *Example showing the three constituents of LAN-XI Open API on a module's web page*



Licenses installed in this module

BZ-7848-A LAN-XI stand-alone recorder	Permanent
BZ-5951-L-N01 LAN-XI input channel streaming	Permanent
BZ-5952-L-N01 LAN-XI output channel streaming	Permanent

Add new license

License key(s):

1.3 What is Open API for LAN-XI?

Open API is a wire protocol based on REST and JSON – technologies known from the Web. This allows you to use it from your own programs.

The control and setup part of Open API is a wire protocol on top of HTTP. This means that any modern operating system and language can be used. Through HTTP commands (GET, PUT and POST) you control settings in the module and set up TCP sockets that stream data to and from your program. Since the setup protocol is based on HTTP, the necessary libraries come with your favourite programming language, such as C#, Objective C, Python or C/C++.

Another advantage is that most programmers already understand the basic "idioms". See the table below, which also shows the similarity to the CRUD idioms in database programming.

HTTP Method	CRUD (database)	Description
`POST`	**C**reate	Create a new resource (at parent-to-be) or perform action
`GET`	**R**etrieve	Retrieve a representation of a resource without side effects
`PUT`	**U**pdate	Update a resource
`DELETE`	**D**elete	Delete a resource

Streaming data from (or to) a LAN-XI module is a bit more complicated. At github.com/hbk-world you can find sample code written in C# and Python. The samples demonstrate how to interpret the binary stream. If you prefer another programming language, we suggest that you download Kaitai Struct (available at github.com/hbk-world), and generate a parser for the streaming format in your chosen programming language.

Also available at github.com/hbk-world are the latest firmware for LAN-XI as well as a draft of the latest version of the programmer's reference.

In the following pages we will use two very nice third-party tools:

- Wireshark® (wireshark.org)

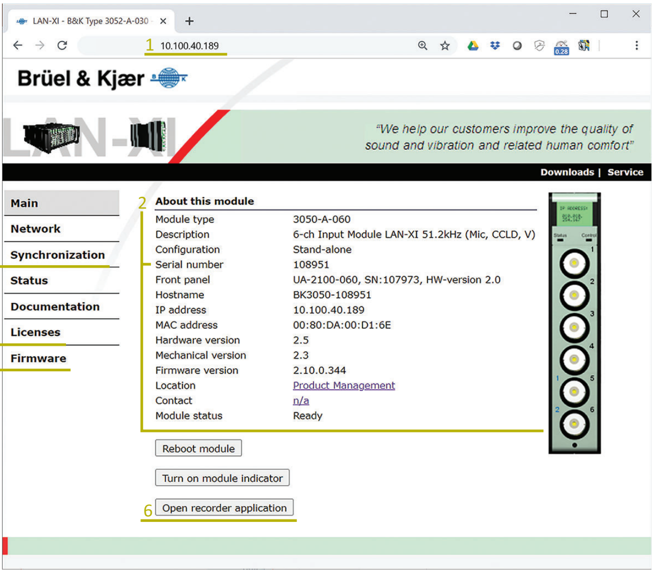
  We use Wireshark to monitor the traffic on the cable. This is indispensable when debugging.
- Postman (postman.com)

  We use Postman to generate test commands. An alternative is cURL.

1.4 The LAN-XI Module Home Page

LAN-XI modules come with an embedded web server, or home page, which provides a lot of relevant information.

Fig. 1.2 Example of a LAN-XI module's home page.



About this module

Module type	3050-A-060
Description	6-ch Input Module LAN-XI 51.2kHz (Mic, CCLD, V)
Configuration	Stand-alone
Serial number	108951
Front panel	UA-2100-060, SN:107973, HW-version 2.0
Hostname	BK3050-108951
IP address	10.100.40.189
MAC address	00:80:DA:00:D1:6E
Hardware version	2.5
Mechanical version	2.3
Firmware version	2.10.0.344
Location	Product Management
Contact	n/a
Module status	Ready

1) **IP adddress**: Found in the address bar of the browser and under "About this module"

2) **About this module**: Serial number, front-panel image, etc.

3) **Synchronization**: Gives information about PTP status

4) **Licenses**: For adding licenses that you have received by, for example, email

5) **Firmware**: Update the module's firmware. Alternatively, you can update a multi-module system in one operation using the Front-end Setup PC application.

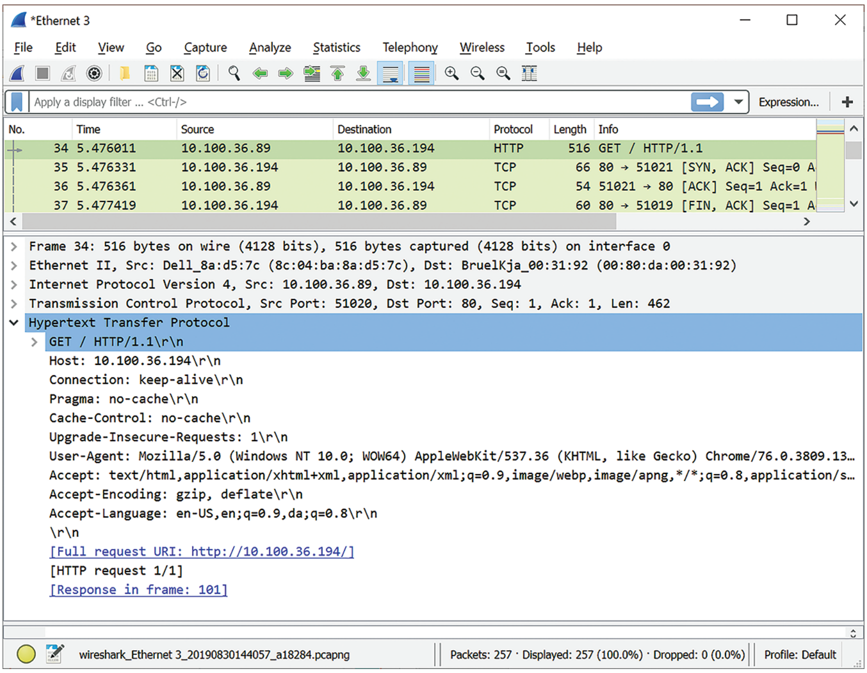
6) Open recorder application: Starts a web application known as LAN-XI Notar, which is the original foundation for Open API. LAN-XI Notar is described as the "Recorder" in this guide. You can

try out the Recorder and even use Wireshark to see what goes on behind the scenes.

Like the setup protocol, the home page is (by definition) based on HTTP. This makes it a good place to start understanding the REST protocol, while digging into the functionality of the support we get from the home page.

Fig. 1.3 shows how the retrieval of the home page looks in Wireshark. The top window of Wireshark shows each packet as a single line. Here we have selected packet 34, which is then decoded in the window below. Often there is a third window below this, which shows the same in hexadecimal notation. It is not shown here.

Fig. 1.3 `GET` *of home page, shown in Wireshark*



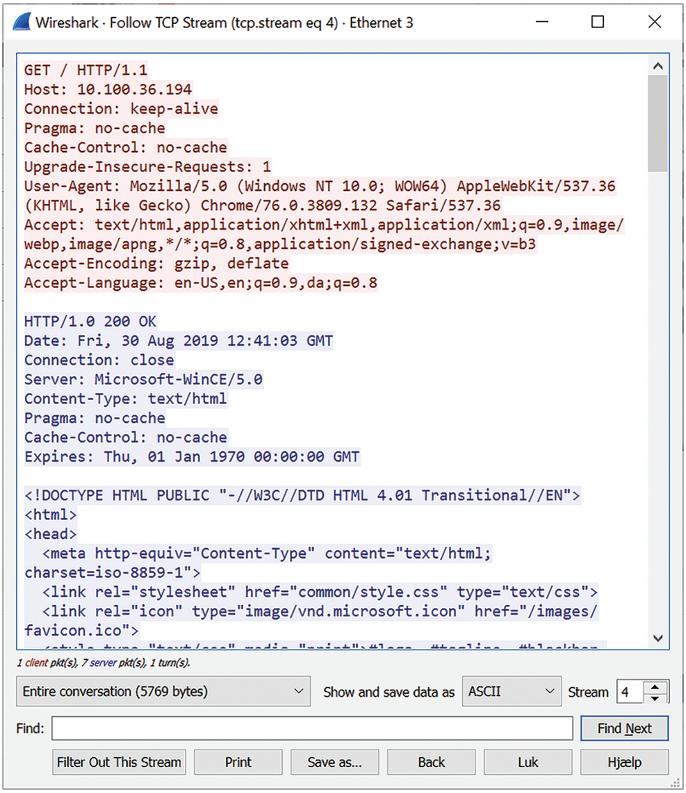
```
No.   Time      Source          Destination     Protocol  Length  Info
  34  5.476011  10.100.36.89    10.100.36.194   HTTP         516  GET / HTTP/1.1
  35  5.476331  10.100.36.194   10.100.36.89    TCP           66  80 → 51021 [SYN, ACK] Seq=0 A
  36  5.476361  10.100.36.89    10.100.36.194   TCP           54  51021 → 80 [ACK] Seq=1 Ack=1 
  37  5.477419  10.100.36.194   10.100.36.89    TCP           60  80 → 51019 [FIN, ACK] Seq=1 A
```

```
> Frame 34: 516 bytes on wire (4128 bits), 516 bytes captured (4128 bits) on interface 0
> Ethernet II, Src: Dell_8a:d5:7c (8c:04:ba:8a:d5:7c), Dst: BruelKja_00:31:92 (00:80:da:00:31:92)
> Internet Protocol Version 4, Src: 10.100.36.89, Dst: 10.100.36.194
> Transmission Control Protocol, Src Port: 51020, Dst Port: 80, Seq: 1, Ack: 1, Len: 462
v Hypertext Transfer Protocol
  > GET / HTTP/1.1\r\n
    Host: 10.100.36.194\r\n
    Connection: keep-alive\r\n
    Pragma: no-cache\r\n
    Cache-Control: no-cache\r\n
    Upgrade-Insecure-Requests: 1\r\n
    User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/76.0.3809.13…
    Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8,application/s…
    Accept-Encoding: gzip, deflate\r\n
    Accept-Language: en-US,en;q=0.9,da;q=0.8\r\n
    \r\n
    [Full request URI: http://10.100.36.194/]
    [HTTP request 1/1]
    [Response in frame: 101]
```

```
wireshark_Ethernet 3_20190830144057_a18284.pcapng    Packets: 257 · Displayed: 257 (100.0%) · Dropped: 0 (0.0%)    Profile: Default
```

When you right-click on packet 34 in the top window and select **Follow TCP Stream**, you get the dialog shown in Fig. 1.4. This shows all of the packets in the HTTP conversation, starting with packet 34. The HTTP **request** is shown in red, while the **response** is shown in blue.

Fig. 1.4 *Request and response of a single HTTP command*


Wireshark · Follow TCP Stream (tcp.stream eq 4) · Ethernet 3

```
GET / HTTP/1.1
Host: 10.100.36.194
Connection: keep-alive
Pragma: no-cache
Cache-Control: no-cache
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/76.0.3809.132 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/
webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9,da;q=0.8

HTTP/1.0 200 OK
Date: Fri, 30 Aug 2019 12:41:03 GMT
Connection: close
Server: Microsoft-WinCE/5.0
Content-Type: text/html
Pragma: no-cache
Cache-Control: no-cache
Expires: Thu, 01 Jan 1970 00:00:00 GMT

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html;
charset=iso-8859-1">
  <link rel="stylesheet" href="common/style.css" type="text/css">
  <link rel="icon" type="image/vnd.microsoft.icon" href="/images/
favicon.ico">
```

1 client pkt(s), 7 server pkt(s), 1 turn(s).

Entire conversation (5769 bytes) | Show and save data as: ASCII | Stream: 4

Find:

Without going into too much detail we see that HTTP messages contain:

- Two mandatory lines. In the request, the first line contains the command type (`GET`), the HTTP standard used (`1.1`), and the URI of the command (`/`), while in the response we see the error code (`200 OK`). The second mandatory line of the request contains the host part of the URI, which in this case, is given as an IP address.
- Several optional lines follow which control pipelining, etc.

- A double line break (`\r\n\r\n`) marks the end of the header and the start of the body.
- A `GET` request has no body, but the `GET` response does, often with a lot of data (here, in HTML).

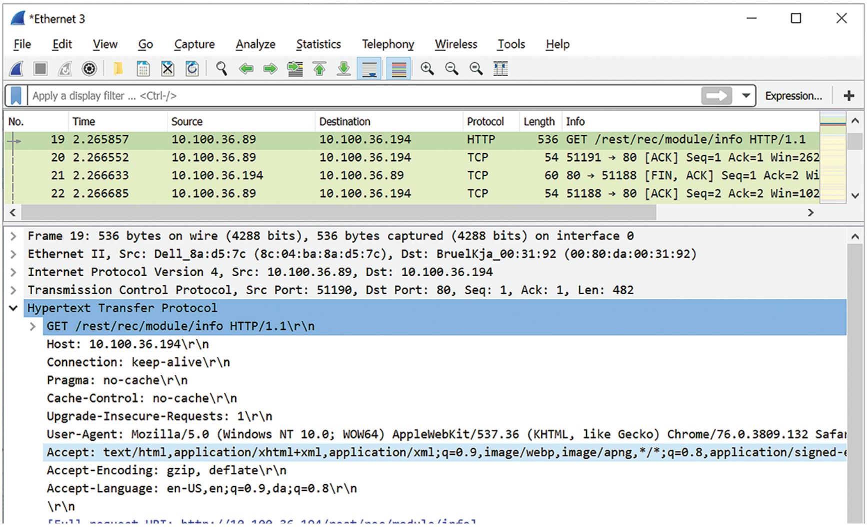
1.5 A REST Command

Instead of using the browser to retrieve web pages containing HTML, we can use it to send a simple command to get the information in JSON format from a LAN-XI module. In the address bar of our browser we write the following (for the same module):

`http://10.100.36.194/rest/rec/module/info/`.

The first part of this URI is the IP address, which must match the module's address, followed by `/rest/rec`, which must always be there to access LAN-XI Open API. After this we move into the module's hierarchical object model. In this case we access the `module` object and underneath that, the `info` object. Fig. 1.5 shows the conversation in Wireshark.

Fig. 1.5 *Request for the* `module/info` *object, shown in Wireshark*

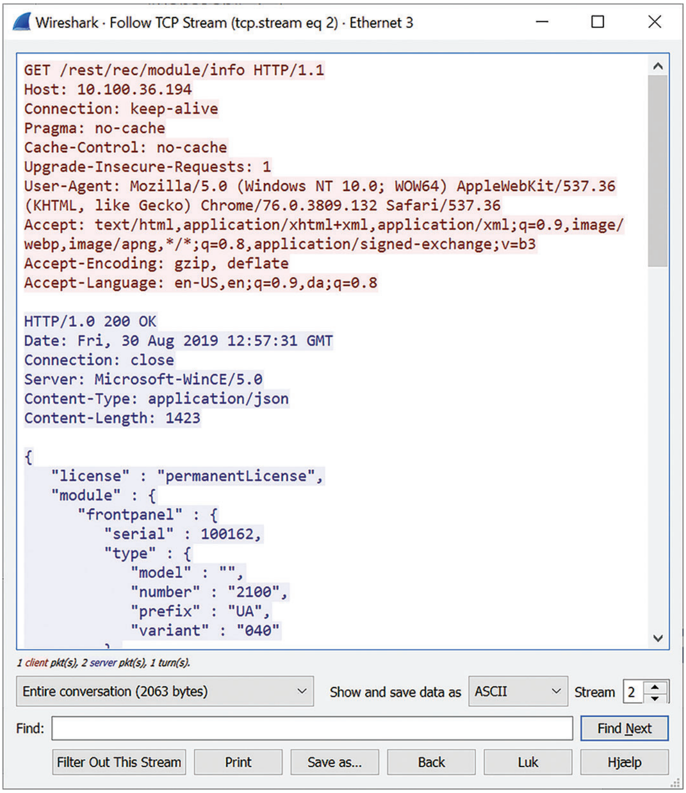


```
No.   Time       Source          Destination     Protocol  Length  Info
  19  2.265857   10.100.36.89    10.100.36.194   HTTP      536     GET /rest/rec/module/info HTTP/1.1
  20  2.266552   10.100.36.89    10.100.36.194   TCP        54     51191 → 80 [ACK] Seq=1 Ack=1 Win=262
  21  2.266633   10.100.36.194   10.100.36.89    TCP        60     80 → 51188 [FIN, ACK] Seq=1 Ack=2 Wi
  22  2.266685   10.100.36.89    10.100.36.194   TCP        54     51188 → 80 [ACK] Seq=2 Ack=2 Win=102
```

```
> Frame 19: 536 bytes on wire (4288 bits), 536 bytes captured (4288 bits) on interface 0
> Ethernet II, Src: Dell_8a:d5:7c (8c:04:ba:8a:d5:7c), Dst: BruelKja_00:31:92 (00:80:da:00:31:92)
> Internet Protocol Version 4, Src: 10.100.36.89, Dst: 10.100.36.194
> Transmission Control Protocol, Src Port: 51190, Dst Port: 80, Seq: 1, Ack: 1, Len: 482
v Hypertext Transfer Protocol
  > GET /rest/rec/module/info HTTP/1.1\r\n
    Host: 10.100.36.194\r\n
    Connection: keep-alive\r\n
    Pragma: no-cache\r\n
    Cache-Control: no-cache\r\n
    Upgrade-Insecure-Requests: 1\r\n
    User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/76.0.3809.132 Safar
    Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8,application/signed-e
    Accept-Encoding: gzip, deflate\r\n
    Accept-Language: en-US,en;q=0.9,da;q=0.8\r\n
    \r\n
    [Full request URI: http://10.100.36.194/rest/rec/module/info]
```

Again, we right-click the start of the HTTP conversation (now packet 19) and select **Follow TCP stream**. The resulting dialog is found in Fig. 1.6. Note that the response this time says the content is in JSON format (`Content-Type: application/json`) with a length of 1423 (`Content-Length: 1423`). This makes it relatively easy for us to parse the JSON in our code, providing us with the relevant setup information.

Fig. 1.6 *HTTP conversation with JSON data*



```
GET /rest/rec/module/info HTTP/1.1
Host: 10.100.36.194
Connection: keep-alive
Pragma: no-cache
Cache-Control: no-cache
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/76.0.3809.132 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/
webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9,da;q=0.8

HTTP/1.0 200 OK
Date: Fri, 30 Aug 2019 12:57:31 GMT
Connection: close
Server: Microsoft-WinCE/5.0
Content-Type: application/json
Content-Length: 1423

{
   "license" : "permanentLicense",
   "module" : {
      "frontpanel" : {
         "serial" : 100162,
         "type" : {
            "model" : "",
            "number" : "2100",
            "prefix" : "UA",
            "variant" : "040"
```

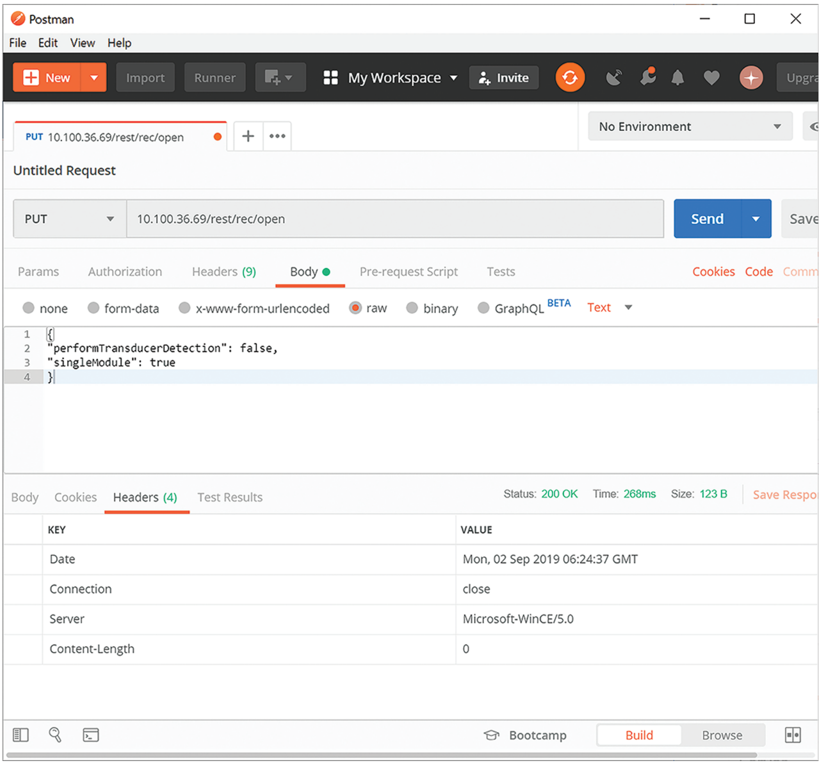
Wireshark · Follow TCP Stream (tcp.stream eq 2) · Ethernet 3
1 client pkt(s), 2 server pkt(s), 1 turn(s).
Entire conversation (2063 bytes)
Show and save data as
ASCII
Stream
2
Find:
Find Next
Filter Out This Stream
Print
Save as...
Back
Luk
Hjælp

Using the browser's address bar for writing `GET` commands is fine, because that is all an address bar does. It implicitly always sends `GET` commands with the given URI and these do not contain a body.

However, when you need to send PUT commands, the browser cannot so easily help you. This might be where you start writing your program, but if you like to handcraft tests a little bit further, you may use Postman, which is a third-party program with a large community. Fig. 1.7 shows a PUT command sent via Postman. Note that PUT is selected in the drop-down box next to the IP address of the module. Postman automatically creates good default headers. Below this we have written four lines of JSON, which Postman will use in the body of the request. Further below we have the response – here we have chosen to look at the headers of this, since in this case, the body is empty.

Postman is clever. You can, for example, set the IP address as a global variable so that you can save scripts that may work on any module.

Fig. 1.7 *PUT command is sent via Postman*


Postman
File Edit View Help
New
Import
Runner
My Workspace
Invite
Upgra
PUT 10.100.36.69/rest/rec/open
No Environment
Untitled Request
PUT
10.100.36.69/rest/rec/open
Send
Save
Params Authorization Headers (9) Body Pre-request Script Tests
Cookies Code Comm
none form-data x-www-form-urlencoded raw binary GraphQL BETA Text
Body Cookies Headers (4) Test Results
Status: 200 OK Time: 268ms Size: 123 B
Save Respo
Bootcamp
Build
Browse

```
{
"performTransducerDetection": false,
"singleModule": true
}
```

KEY	VALUE
Date	Mon, 02 Sep 2019 06:24:37 GMT
Connection	close
Server	Microsoft-WinCE/5.0
Content-Length	0

Please note that when a LAN-XI module is controlled by Open API, the display still shows correct status. You can even use the small button next to the display to start/stop your ongoing measurement, etc.

1.6 Streaming

If you look at a typical application, the streaming of data takes 99% or more of the bandwidth on the wire. For the sake of performance this is binary. You can find samples on how to parse the binary header data and process the sample data at github.com/hbk-world. There are some samples in C#, others in Python. In both cases we use the HTTP/web libraries that come with the language.

A typical flow could be:

1) Connect to the LAN-XI module and put it in Recorder mode (Open API mode).

   Optionally, do a TEDS detection to learn about transducer sensitivity, etc.

2) Get or set the sample rate and filters.

3) Tell the module to stream data to your client. By default, data goes to an inserted SD card.

4) Create a socket and start receiving blocks of data.

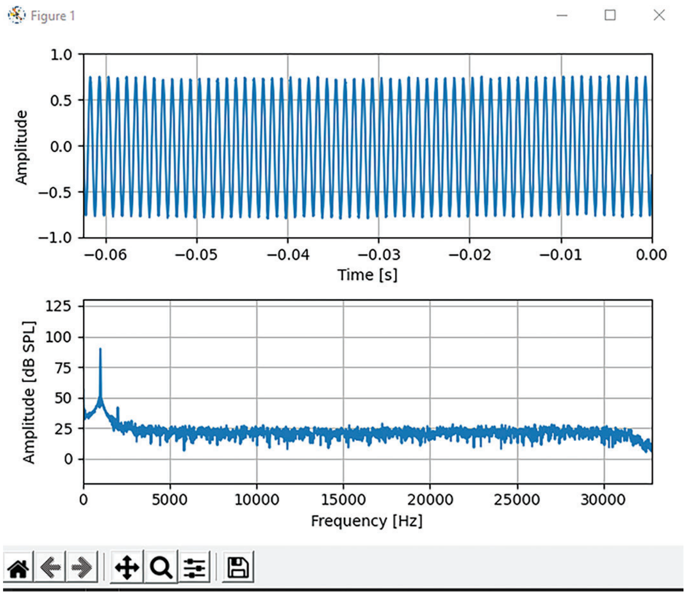
5) Parse the header to find metadata and description of channels, including scale factors.

6) Buffer samples and display, filter and/or store as needed.

7) Repeat steps 6 and 7 per block of data. (Note: This might require a threaded implementation.)

Fig. 1.8 shows the output from a Python sample named RealtimePlot.py, available at github.com/hbk-world/LAN-XI-Open-API-python-examples. This sample is animated and uses threaded code. The top graph is in the time domain and shows a 1 kHz sine (simply generated with a phone), while the bottom graph shows an FFT of the signal (using a Hamming window). In this case a microphone is connected, and the Y-axis shows magnitude in dB re 20 µPa (based on TEDS sensitivity).

Fig. 1.8 *Output from Python sample program analyzing a 1 kHz tone from a phone*



If the LAN-XI module has one or more generators, you can also control these. Each generator has two specific modes:

- A classic signal generator with two internal sources. You can, for example, select one source to be a fixed sine, and the other to be a swept sine. The generator will create the two signals, mix them internally, and output the summed signal on the front connector of the module. This will go on until stopped.
- In the same way that you can stream data from a module, you can stream binary data to the generator in the module, and it will output the analog data on the module's front connector.

Earlier we saw that the HTTP-based commands are easy to see in Wireshark. If you want to use this tool with the binary data streams, you will appreciate the "Wireshark Dissector" found at github.com/hbk-world/LAN-XI-Open-API-Tools.

1.7 Ground Rules

When using LAN-XI Open API there are rules that must be respected:

- LAN-XI supports both Open API and BK Connect/PULSE LabShop, but only one at a time.

  You should, for example, power cycle LAN-XI between sessions with the Open API and other clients.
- Multiple Open API clients directed towards the same LAN-XI module are possible at the same time, but the LAN-XI module can only stream to a single client or to the SD card.

  Other clients may show status and allow users to start or stop measurements.
- HTTP-based commands, by definition, operate on a single module. However, it is possible to arm all modules, and then start sample-synchronous sampling – see the code samples.
- LAN-XI supports neither HTTPS nor WebSocket streams and has not been verified with IPv6.

  You can, however, select to have one socket per channel, instead of the default which sends all data on a single socket.
- PTP synchronization requires that modules and frames can “see” each other via the left connector on frames (rear view). Thus, if you have only two frames, or a frame and a stand-alone module, you can connect your PC to the right connector (rear view) on the frame. Larger setups will require a PTP-aware switch to which the PC can also be connected. Note that some low-quality switches may support PTP, but still deliver poor timing. HBK recommends our UL-0265 or high-quality brands like Hirschmann™ and Cisco®.

Chapter 2

Recorder

For historic reasons, LAN-XI Open API is built on top of the LAN-XI Notar application (described as the “Recorder” in this guide). This has defined the various states that the application can – and must – go through.

The Recorder on LAN-XI can be used to record a signal to a network stream or SD card – and generate a signal on modules with output. All command paths related to the Recorder are prepended by: `/rest/rec/`.

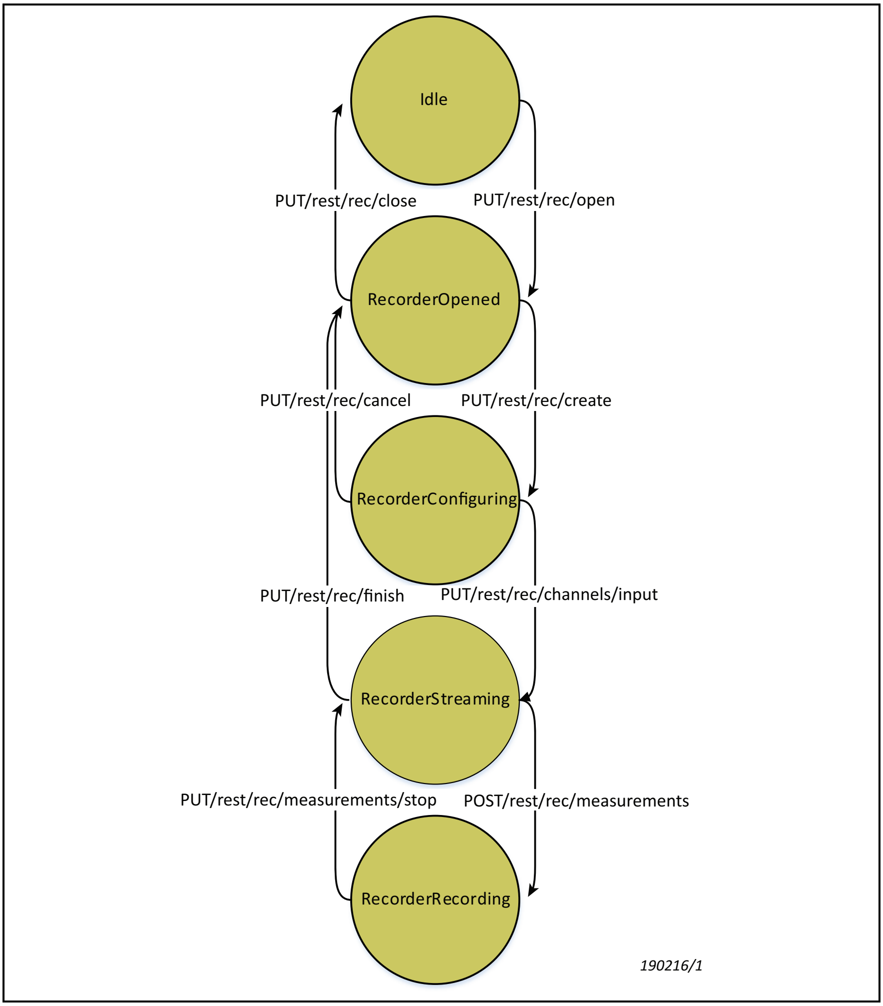
The commands are case-insensitive, but the JSON body is case-sensitive.

The application supports synchronization and alignment of samples across several LAN-XI modules, but can also be used on a single module.

2.1 State Machine

The Recorder operates using several state domains. The main states are the module states. The module states relevant to the Recorder are illustrated in Fig. 2.1.

Fig. 2.1 *Module states relevant to the Recorder – and transitions between the states*



When using PTP synchronization, two other states are important to monitor: Input status and PTP status.

2.2 Generic Flow

Action	REST Command	Resulting Module State
Post-boot state		Idle
Open Recorder	`/rec/open`	RecorderOpened
Create configuration	`/rec/create`	RecorderConfiguring
Configure input channels	`/rec/channels/input`	RecorderStreaming
Get streaming port	`/rec/destination/socket` or `/rec/destination/sockets`	
Open Socket Connection(s)		
Start streaming	`/rec/measurements`	RecorderRecording
Fetch Samples		
End streaming	`/rec/measurements/stop`	RecorderStreaming
Finish recording session	`/rec/finish`	RecorderOpened
Close Recorder	`/rec/close`	Idle

It is not necessary to perform the entire chain of actions for each recording. If no configuration changes are necessary, it is sufficient to end streaming with `/rec/measurements/stop` and restart it with `/rec/measurements/start`.

When streaming data in multi-socket mode, all sockets must be connected before streaming is started, or the module will indicate an error. Once streaming is started, an overrun condition or a broken connection on some streams will not affect other streams, which will continue transferring data.

2.3 Managing Stored Recordings

Recordings stored on the SD card can be listed, downloaded, and deleted.

2.3.1 Obtaining a List of Recordings

Returns a list of recordings stored on the SD card.

URI:	http://<ip>/rest/rec/measurements
Supported Methods:	`GET`
Valid States:	All
Resulting State:	Unchanged
Body:	None

Example response:

```
[
   {
      "size": 9446395,
      "duration": 4000,
      "uri": "/rest/rec/measurements/20210126-053627(UTC)---0000074254",
      "setup":
      {
         "channels" : [ … ],
         "datetime" : 1611639387689,
         "fileFormat" : "bkc",
         "maxSize" : 2147483648,
         "module" : { … },
         "name" : "Rec3",
         "recordingMode" : "Single"
      }
   },
   … (more recordings) …
]
```

size	integer	The size, in bytes, of the recording
duration	integer	The length, in milliseconds, of the recording
uri	string	A URI used to download, delete, or otherwise reference the recording
setup	object	The measurement setup used to produce the recording. This is identical to the setup applied to `/rest/rec/channels/input` and contains the name and date/time of the recording

2.3.2 Downloading a Recording

Downloads a recording stored on the SD card.

URI:	http://<ip><uri>
Supported Methods:	`GET`
Valid States:	RecorderOpened, RecorderConfiguring, RecorderStreaming
Resulting State:	RecorderDownloading
Body:	None
Parameters (optional)	format=bkc24

To specify the recording of interest, append the <uri> from the list of recordings obtained from a `GET` or `POST` request to `/rest/rec/measurements`.

The response will be a WAV or BKC file, depending on the file format selected at time of recording.

BKC files are downloaded in 32-bit format by default. This format is supported by all versions of BK Connect, but takes up more space than strictly necessary.

If using a recent version of BK Connect that supports 24-bit BKC then it is possible to reduce bandwidth and storage requirements by appending "?format=bkc24" to download a 24-bit BKC file that is, `GET http://<ip><uri>?format=bkc24`.

2.3.3 Deleting a Recording

Removes a recording from the SD card.

URI:	http://<ip><uri>
Supported Methods:	`DELETE`
Valid States:	All
Resulting State:	Unchanged
Body:	None

To specify the recording to be deleted, append the <uri> from the list of recordings obtained from a `GET` or `POST` request to `/rest/rec/measurements`.

2.4 Input Related Commands

2.4.1 /rest/rec/open

This command opens the Recorder application on the module.

URI:	http://<ip>/rest/rec/open
Supported Methods:	`PUT`
Valid States:	Idle
Resulting State:	RecorderOpened
Body:	Optional

Body (optional):

```
{
   "performTransducerDetection": true,
   "singleModule": true
}
```

Send to slaves first, then the master.

performTransducerDetection	boolean	If set to *true*, the module will automatically perform TEDS transducer detection when the Recorder is opened. Otherwise, initial transducer detection is skipped. Skipping transducer detection reduces the time the Recorder application takes to open. If not specified, this parameter defaults to *true*.

singleModule	boolean	If set to *true*, the Recorder application will open in single-module mode. This means the module will be configured to run as a single, independent module. Otherwise, the Recorder application will open in multi-module mode. This means the module will prepare to run as part of a larger, sample-synchronous, multi-module configuration. Set this parameter depending on the desired recording configuration. If not specified, this parameter defaults to *true*.

2.4.2 /rest/rec/close

This command closes the Recorder on the module.

URI:	http://<ip>/rest/rec/close
Supported Methods:	`PUT`
Valid States:	RecorderOpened
Resulting State:	Idle
Body:	None

2.4.3 /rest/rec/create

This command creates a recording configuration for the Recorder.

URI:	http://<ip>/rest/rec/create
Supported Methods:	`PUT`
Valid States:	RecorderOpened
Resulting State:	RecorderConfiguring
Body:	None

2.4.4 /rest/rec/cancel

This command cancels and discards the current configuration.

URI:	http://<ip>/rest/rec/cancel
Supported Methods:	`PUT`
Valid States:	RecorderConfiguring
Resulting State:	RecorderOpened
Body:	None

2.4.5 /rest/rec/finish

This command ends the current recording session, returning the Recorder to the opened state.

URI:	http://<ip>/rest/rec/finish
Supported Methods:	`PUT`
Valid States:	RecorderStreaming
Resulting State:	RecorderOpened
Body:	None

2.4.6 /rest/rec/module/time

This command sets the system date and time in the module. It will not affect the PTP time.

The module will ignore this command, if its system time already is set with `year >= 2009`.

URI:	http://<ip>/rest/rec/module/time
Parameters:	One parameter that is an ASCII string specifying the number of milliseconds since 1/1-1970 0:00:00 (UTC), including leap seconds
Supported Methods:	`PUT`
Valid States:	

Body example (plain text):

```
1388534400000
```

1/1-2014 0:00:00 (UTC) used in example: 1000 msec/sec × 60 sec/min × 60 min/hr × 24 hr/day × 16071 days (between 1/1-1970 and 1/1-2014) = *1388534400000*

2.4.7 /rest/rec/module/info

This command returns information about this module.

URI:	http://<ip>/rest/rec/module/info
Supported Methods:	`GET`
Valid States:	All

Body example from LAN-XI Bridge Module Type 3057 (JSON):

Please note: Depending on the module, not all fields may be presented.

```
{
   "license" : "permanentLicense",
   "module" : {
      "frontpanel" : {
         "serial" : 105002,
         "type" : {
            "model" : "",
            "number" : "2121",
            "prefix" : "UA",
            "variant" : "030"
         },
         "version" : {
            "hardware" : "2.0.0.0"
         }
      },
      "serial" : 105224,
      "type" : {
         "model" : "B",
         "number" : "3057",
         "prefix" : "",
         "variant" : "030"
      },
      "version" : {
         "firmware" : "2.7.2669.6",
         "hardware" : "1.3.0.0"
      }
```

```
    },
    "moduleState" : "Idle",
    "numberOfInputChannels" : 3,
    "numberOfOutputChannels" : 0,
    "preampInputSupported" : false,
    "sdCardInserted" : false,
    "supportedBridgeCompletion" : [ "None", "Half", "QuarterPos" ],
    "supportedBridgeSupply" : [ "DC Voltage", "DC Current" ],
    "supportedConnectors" : [ "BNC", "Charge", "Bridge", "Diff Charge" ],
    "supportedFilters" : [ "DC", "0.1 Hz 10%", "0.7 Hz", "1.0 Hz 10%", "7.0 Hz", "22.4 Hz" ],
    "supportedMaxBridgeCurrent" : 0.0250,
    "supportedMaxBridgeVoltage" : 10.0,
    "supportedOutputRanges" : [],
    "supportedQuarterCompletionImpedance" : [ "120 Ohm", "350 Ohm", "1 kOhm" ],
    "supportedRanges" : [ "0.316 Vpeak", "10 Vpeak" ],
    "supportedRemoteSenseWiring" : [ "None", "SinglePos", "Double" ],
    "supportedSampleRates" : [
        262144,
        131072,
        65536,
        32768,
        16384,
        8192,
        4096,
        2048,
        1024,
        512,
        256,
        128
    ]
}
```

Most of the above properties are described in section 2.4.8.1. The rest are described in the following table.

license	string	The kind of license currently in use *{"permanentLicense", "noLicense"}*
module	object	Describes the module: type/serial no., firmware version and front panel details
moduleState	string	Current state of the module's state machine. See section 2.1.

numberOfInputChannels	integer	Number of input channels available in the module
numberOfOutputChannels	integer	Number of output channels available in the module
preampInputSupported	boolean	Whether or not a CCLD preamplifier is available
sdCardInserted	boolean	Whether or not an SD card is inserted in the slot on the back of the module
supportedFilters	array of strings	The basic low-pass filters available, presented in an array
supportedOutputRanges	array of strings	Output ranges available, presented in an array
supportedRanges	array of strings	Input ranges available, presented in an array
supportedSampleRates	array of integers	Sample rates supported by the module, presented in an array*
syncModeChangeSupported	boolean	

* It is possible for a home-made application via Open API to control independent modules with different sample rates, even if they are in the same C- or D-frame. However, PULSE LabShop and BK Connect require the entire LAN-XI system to have a common sample rate.

The table below shows some other properties supported by the LAN-XI CAN module and not shown in the above example.

supportedHatsChannelPairs	array of arrays	An array where each element is an array containing a pair of channels supporting HATS (AES) input (for example, *[[3,7], [4,8]]*). This property is not shown in the above example and currently only supported by the LAN-XI CAN module.
numberOfChannels	integer	Number of CAN channels supported by the module
supportedLowSpeedChannels	array of integers	An array of integers containing channel numbers supporting low-speed CAN bus type

supportedHighSpeedChannels	array of integers	An array of integers containing channel numbers supporting high-speed CAN bus type
supportedObd2Channels	array of integers	An array of integers containing channel numbers supporting OBD-II
supportedLowSpeedBaudrates	object	Supported baud rate range for low-speed bus type (for example, *{“auto”: 0, “min”: 10000, “max”: 125000 }*)
supportedHighSpeedBaudrates	object	Supported baud rate range for high-speed bus type (for example, *{“auto”: 0, “min”: 10000, “max”: 1000000 }*)
supportedModes	array of strings	An array of strings describing supported modes (for example, *[“Passive”, “Active”, “Write”]*)
supportedBusTypes	array of strings	An array of strings describing the supported bus type(s) (for example, *[“LowSpeed”, “HighSpeedWithoutTerminator”, “HighSpeedWithTerminator”]*)

2.4.8 Input Channel Configuration

The current input channel configuration can be set and retrieved, and the module stores a default setup that can be retrieved.

2.4.8.1 Data Exchange

For all input channel commands, the configuration is transferred in the body of the request. JSON is used to describe the channel setup, and the following (Type 3057 sample) layout is used:

```
{
   "channels" : [
      {
         "bandwidth" : "102.4 kHz",
         "bridgeCompletion" : "None",
         "bridgeQuarterCompletionImpedance" : "350 Ohm",
         "bridgeExcCurrent" : 0.0,
         "bridgeExcOn" : false,
         "bridgeExcVoltage" : 0.0,
         "bridgeRemoteSenseWiring" : "None",
         "bridgeShunt" : false,
         "bridgeSingleEnd" : false,
         "bridgeSupplyType" : "DC Voltage",
         "ccld" : false,
         "hats" : false,                          // Supported by CAN modules
         "channel" : 1,
         “ConnectorSelect” : “BNC”,
         "destinations" : [ "sd" ],
         "enabled" : true,
         "filter" : "7.0 Hz",
         "floating" : false,
         "name" : "Channel 1",
         "polVolt" : false,
         "range" : "10 Vpeak",
         "transducer" : {
            "requires200V" : false,
            "requiresCcld" : false,
            "sensitivity" : 1,
            "serialNumber" : 0,
            "type" : {
               "model" : "",
               "number" : "None",
               "prefix" : "",
               "variant" : ""
            },
            "unit" : "V"
         }
      },
… // further channels omitted
   ],
   “canChannels” : [
               {
               “channel” : 1,
               “name” : “CAN 1”,
               “enabled” : true,
               “destinations” : [“socket”],
               “baudRate” : 0,
               “baudRateDetectionTimeout” : 300,
                “mode” : “Passive”,
                “busType” : “HighSpeedWithTerminator”,
                “loopback” : false
            },
            {
            …                    // Next CAN channel (not shown)
            }
    ],

    "maxSize" : 2147483647,
    "name" : "Default setup",
    "recordingMode" : "Single",
    “fileFormat”: “wav”
}
```

The properties are as follows:

channels	object	An array of channel and transducer parameters where each element is an object which consists of the parameters below.
bandwidth	string	The bandwidth of the channel (sample rate will be 2.56 times this number) *{“50 Hz”, “100 Hz”, “200 Hz”, “400 Hz”, “800 Hz”, “1.6 kHz”, “3.2 kHz”, “6.4 kHz”, “12.8 kHz”, “25.6 kHz”, “51.2 kHz”, “102.4 kHz”, “204.8 kHz”}* Maximum depends on module type. The same bandwidth must be used for all channels.
bridgeCompletion	string	*{“None”, “Half”, “QuarterPos”}* *None* means no completion. Type 3057 supports quarter-bridge completion in positive arm only.
bridgeQuarter CompletionImpedance	string	*{“Off”, “120 Ohm”, “350 Ohm”, “1 kOhm”}* *Off* is automatically set if ***bridgeCompletion*** is *None*.

bridgeExcCurrent	decimal	*{0.000-0.0250}* [Ampere] Ignored if ***bridgeSupplyType*** is *DC Voltage*
bridgeExcOn	boolean	*{true, false}* Excitation must be on for bridge measurements. **Caveat:** In current firmware, this MUST be the same for all channels
bridgeExcVoltage	decimal	*{0.0 – 10.0}* [Volt] Ignored if ***bridgeSupplyType*** is *DC Current*
bridgeRemoteSense Wiring	string	*{"None", "SinglePos", "Double"}* • *None* = No compensation for voltage drop over cables • *Single* = Compensation calculated by module at measurement start • *Double* = Analogue hardware compensation
bridgeShunt	boolean	*{true, false}* Tells Type 3057 that the user has mounted a shunt*
bridgeSingleEnd	boolean	*{true, false}* Must be *false* when measuring with bridge
bridgeSupplyType	string	*{"DC Voltage", "DC Current"}*
ccld	boolean	Enable/disable the CCLD power supply.
channel	integer	The channel number (*1* is the first channel)
ConnectorSelect	string	*{"BNC", "LEMO", "Charge", "Bridge", "Diff Charge", "T-insert", "T-insert ref"}*†
destinations	array of strings	List of destinations for this channel. Specify *"sd"*, *"socket"* or *"multiSocket"*. (Currently only one destination is supported for a channel, and the same destination must be used for all channels.)

enabled	boolean	Whether or not to include this channel in the recording
filter	string	High-pass filter *{"DC", "0.7 Hz", "7.0 Hz", "22.4 Hz", "Intensity"}*
floating	boolean	Whether the channel should be grounded (*false*) or floating (*true*)
name	string	Name of the channel. This will be included in the WAV file metadata
polVolt	boolean	Enable or disable 200 V polarization voltage. This setting must be the same for all channels.
range	string	Input range *{"0.316 Vpeak","1 Vpeak","10 Vpeak", "31.6 Vpeak"}*
transducer	object	Transducer setup
maxSize	integer	The maximum allowed size of the recording in bytes
name	string	Specifies the name of the recording in UTF-8 encoded Unicode
recordingMode	string	*{"Single", "Semi-continuous"}*
fileFormat	string	Specifies the format of the recording file. Options are: • *"wav"* to produce a recording in wave format • *"bkc"* to generate a BKC file. BKC is the Brüel & Kjaer common file format This property is optional. If the REST client does not specify a file format, the recording will be stored in the wave format.

* Type 3057 contains a fixed 50 kΩ shunt calibration resistor that can be switched on and off as needed. The resistor is connected to two separate pins in the input connector and can thus be connected across any of the four arms in the bridge, depending on the desired sign of the simulated strain. ***bridgeShunt*** controls the internal shunt switch, however the calibration measurement must be handled by the client application.

† The *T-insert* and *T-insert ref* options of the ***ConnectorSelect*** parameter are special charge input variants used to measure transducer capacitance. The capacitance determination requires two measurements, one in each setting. At the time of this writing, this is only supported via the UA-3122-030 front panel.

The next table shows parameters specific to the CAN module. They are ignored by non-CAN modules and may be omitted.

hats	boolean	Enables using AES/HATS (head and torso simulator) inputs. It is supported by CAN modules and requires two input channels (called pair-channels). Pair-channel info can be acquired with the `/rest/rec/module/info` command
canChannels	object	An array of CAN channel parameters, each element being an object consisting of the parameters below
channel	integer	CAN channel number
name	string	A descriptive name for the CAN channel defined by user.
enabled	boolean	Enables (*true*) or disables (*false*) CAN channel.
destinations	array of strings	List of destinations for CAN channel. Specify *“sd”*, *“socket”* or *“multiSocket”*. (Currently only one destination is supported for a channel, and the same destination must be used for all channels.)
baudRate	integer	CAN bus baud rate setup in Bd Valid range depends on bus type as follows: • Low speed: *10000 – 125000* • High speed: *10000 – 1000000* Set the baud rate to *0* for automatic baud rate detection.
baudRateDetectionTimeout	integer	Automatic baud rate detection time out from *1* to *300* seconds. It is ignored if automatic detection is not selected.
mode	string	Communication mode *{“Passive”, “Active”, “Write”}*

busType	string	Selects one of the following supported CAN bus types: *{“LowSpeed”, “HighSpeedWithoutTerminator”, “HighSpeedWithTerminator”}.*
loopback	boolean	For troubleshooting purposes, a copy of sent messages may be received if enabled.

Please note: Run `/rest/rec/module/info` to see complete info, layout and exact value ranges for a given module.

2.4.8.2 /rest/rec/channels/input/default

This command returns the default measurement channel setup for the Recorder.

URI:	http://<ip>/rest/rec/channels/input/default
Supported Methods:	`GET`
Valid States:	All
Body:	See section 2.4.8.1

2.4.8.3 /rest/rec/channels/input

This command returns the current measurement channel setup for the Recorder (when issuing a `GET` request) or does the setup (when issuing a `PUT` request).

URI:	http://<ip>/rest/rec/channels/input
Supported Methods:	`GET`, `PUT`
Valid States:	`PUT`: RecorderConfiguring `GET`: RecorderStreaming
Resulting State:	`PUT`: RecorderStreaming `GET`: State unchanged
Body:	See section 2.4.8.1

2.4.8.4 /rest/rec/channels/cic

This command prepares the desired input channels for CIC (charge injection calibration) measurement and must be sent prior to `/rest/rec/channels/input` command. The selected input channels must be enabled when issuing the input configuration. CIC measurement requires a front panel capable of routing CIC signal to transducers, such as front panels with LEMO connectors. The CIC configuration stays valid until a `/rest/rec/finish` command has been issued.

URI:	http://<ip>/rest/rec/channels/cic
Supported Methods:	GET, PUT
Valid States:	RecorderConfiguring
Resulting State:	(PUT) RecorderConfiguring
Body:	See below

channels	Array of numbers	Array of selected input channel numbers for CIC measurement. The selected channels must be enabled when `/rest/rec/channels/input` issued
generator	Object	Generator object containing following parameters
gain	Number	Generator gain in the range of *0.0* to *0.999999*, which corresponds to full scale
offset	Number	DC offset to be added to generator signal in the range [*−0.999999*, *+0.999999*]
frequency	Number	Generator sine frequency given in Hz
phase	Number	The phase of sine signal given in degrees

Here is an example of CIC configuration:

```
{
   "channels" : [1,2,5,6]
   "generator" :{
      "gain" : 0.9
      "offset" : 0.0
      "frequency" : 1024
      "phase" : 0.0
   }
}
```

It is not possible to use CIC and the generator simultaneously. Therefore, using a generator module like Type 3160 for CIC measurement will reserve its output-1 resources and cannot be used at the same time. However, its second output is still free and can be used.

2.4.8.5 */rest/rec/channels/input/bridgeNulling*

This command returns the current bridge nulling value from the Recorder (when issuing a `GET` request). This is a DC offset in volts, typically representing the value of the bridge before applying/ changing the load. The nulling value survives a power cycle and can be read in any state.

When issuing a `PUT` request, the command will cause an automatic nulling or a reset nulling.

This is much like a scale where you can choose to see the full weight (reset nulling) or let the scale tare out the weight of material that is not interesting (automatic).

URI:	http://<ip>/rest/rec/channels/input/bridgeNulling
Supported Methods:	`GET`, `PUT`
Valid States:	(`PUT`) RecorderStreaming
Resulting State:	(`PUT`) RecorderStreaming
Body:	See section 2.4.8.1

Example (`PUT`):

```
{
  "channels" : [
    {
    "channel" : 1,
    “nulling” : “Automatic”,
    },
... // further channels omitted
  ]
}
```

The properties are as follows:

channels	object	An array of channel and transducer parameters, each element is an object which consists of the parameters below.
channel	integer	*{1..3}*
nulling	string	*{“None”, “Automatic”, “Reset”}*

2.4.8.6 /rest/rec/can/obd2

This command is used to request an OBD-II message and supported by LAN-XI CAN Module Type 3058. The module can store up to 64 OBD-II requests for each channel in an internal list and send them with the desired interval time. The command supports REST `GET` to retrieve OBD-II list, `PUT` to add a new request and `DELETE` to remove a request from the internal list.

URI:	http://<ip>/rest/rec/can/obd2
Supported Methods:	`GET`, `PUT`, `DELETE`
Valid States:	RecorderStreaming
Resulting State:	No change in state
Body:	See example below

Body:

```
{
   “action”: “KeepList”
   "Obd2Messages": [
            {
                     "channel": 2,
                     "messageID": 2016,            // ID = 0x7e0
                     "cycleTime": 250,
                     "messageInfo": 0,
                     "dataSize": 3,
                     "data": [2, 1, 12]            // Request: RPM
            },
            {
                      "channel": 2,
                      "messageID": 2016,              // ID = 0x7e0
                      "cycleTime": 250,
                      "messageInfo": 0,
                      "dataSize": 3,
                      "data": [2, 1, 13]              // Request: Speed
            },
            {
                      "channel": 2,
                      "messageID": 2016,              // ID = 0x7e0
                      "cycleTime": 250,
                      "messageInfo": 0,
                      "dataSize": 3,
                      "data": [2, 1, 17]              // Request: Throttle position
            }
   ]
}
```

As shown, several OBD-II messages can be sent with the same command in order to add to or delete from the internal list. The following table describes OBD-II parameters.

Parameter Name	Options	Type	Description
action	*"KeepList", "Overwrite", "DeleteAll"*	string	Selects the action to perform. • *KeepList*: Keeps the internal list and performs an add or delete operation on it. This is the default action. Supported with both `PUT` and `DELETE` • *Overwrite*: Overwrites the internal list with the new OBD messages in the ***Obd2Messages*** object. Supported with `PUT` only • *DeleteAll*: Deletes all the OBD messages for the selected channel(s). Supported with `DELETE` only See descriptions below.

Obd2Messages		object	An array of OBD-II messages where each element is an object which consists of the parameters below.
channel	*1*, *2*	number	To select CAN channel
messageID	11- or 29-bit ID	number	CAN message ID
cycleTime	*1 – 65535* ms	number	Time in milliseconds between two consecutive messages (defines number of messages per second)
messageInfo	See description	number	This is a bitwise field, with the following meaning: Bit[0]: ExtendedID (*0* = 11-bit, *1* = 29-bit) Bit[1]: RTR (*1* = remote transmission request) Bit[2 – 7]: Reserved **Please note:** When RTR is set, then no data is allowed
dataSize	*0 – 8*	number	Number of data bytes in the next field
data	Up to 8 bytes of data	array of numbers	OBD-II data to be sent to CAN bus

The ***action*** parameter along with the method type (`PUT` or `DELETE`) defines what to do with the internal OBD-II list. If *DeleteAll* action is selected, then the entire OBD-II list for the selected channel is cleared. If *Overwrite* action is selected, then the entire OBD-II list is replaced with the new OBD-II message(s). Use *KeepList* to add or delete one or specific message(s). Some combinations, like *Overwrite* action with `DELETE` method type, are not allowed. See table above for supported methods with the selected action parameter.

Please note: If the mode parameter described in section 2.4.8.1 is not set to *Write* mode, then the OBD-II messages will not be transmitted to CAN bus.

2.4.8.7 /rest/rec/can/sendMessages

This command is used to transmit a number of CAN messages once. The ***mode*** parameter described in section 2.4.8.1 must be set to *Write* mode in order to use this command.

URI:	http://<ip>/rest/rec/can/sendMessages
Supported Methods:	PUT
Valid States:	RecorderRecording
Resulting State:	RecorderRecording

Body:

```
{
  "TxMessages": [
        {
            "channel": 1,
            "messageID": 2016,              // ID = 0x7e0
            "messageInfo": 0,
            "dataSize": 3,
            "data": [2, 1, 12]              // Request: RPM
        },
        {
            "channel": 2,
            "messageID": 2016,              // ID = 0x7e0
            "messageInfo": 0,
            "dataSize": 3,
            "data": [2, 1, 13]              // Request: Speed
        }
  ]
}
```

Command parameters are described in table below.

Parameter Name	Options	Type	Description
TxMessages		object	An array of messages where each element is an object which consists of the parameters below.
channel	*1, 2*	number	To select CAN channel

messageID	11- or 29-bit ID	number	CAN message ID
messageInfo	See description	number	This is a bitwise field, with the following meaning: Bit[0]: ExtendedID (*0* = 11-bit, *1* = 29-bit) Bit[1]: RTR (*1* = remote transmission request) Bit[2 – 7]: Reserved **Please note:** RTR is used to request CAN message from a specific CAN device.
dataSize	0 – 8	number	Number of bytes in the data field
data	Up to 8 bytes of data	array of numbers	Data to be sent to CAN bus. The maximum is 8 bytes.

2.4.9 Detect Commands

2.4.9.1 /rest/rec/channels/input/all/transducers

This command returns information about the transducers connected to the module. The information returned is stored in the module. To perform TEDS detection on connected transducers, refer to the `POST /rest/rec/channels/input/all/transducers/detect` command.

URI:	http://<ip>/rest/rec/channels/input/all/transducers
Supported Methods:	`GET`
Valid States:	All

Body (JSON):

The body returned is a JSON array of *N* objects (which may be ***null*** values), with *N* being the number of input channels available. The objects are arranged by channel number, beginning with the first channel.

A ***null*** value indicates that either nothing is connected to the channel, or no TEDS data is available. Otherwise, an object presents the information gathered.

Example:

```
[
    {
            "direction" : "x",
            "requires200V" : 0,
            "requiresCcld" : 1,
            "sensitivity" : 0.01029344982280505,
            "serialNumber" : 50891,
            "teds" : "EdAaIYJlYwBgpD1kBZ4Zm24NaGNNhADgIBAIBAKBQA==",
            "type" : {
              "model" : "B",
              "number" : "4525",
              "prefix" : "",
              "variant" : "001"
            },
            "unit" : "m/s^2"
    },
    null,
... // further transducers are omitted
]
```

The fields returned for each transducer may vary.

2.4.9.2 /rest/rec/channels/input/all/transducers/detect

This command starts the TEDS detection of transducers. Any TEDS information retrieved is stored and not returned with this call. Use `GET /rest/rec/channels/input/all/transducers` to fetch the data.

URI:	http://<ip>/rest/rec/channels/input/all/transducers/detect
Supported Methods:	`POST`
Valid States:	All
Body:	None

2.4.9.3 */rest/rec/can/detectCables*

This command is used to detect and gather information on cables attached to CAN channels. The module scans both CAN channels simultaneously and returns a JSON object containing the result of scanning.

URI:	http://<ip>/rest/rec/can/detectCables
Supported Methods:	`POST`
Valid States:	All except RecorderRecording
Body:	None

Here is an example of the returned information by this command.

```
{
   "channels": [
      {
         "cable": {
            "BkNo": "ZH-0717",
            "ID": 17,
            "description": "CAN breakout box",
            "lemoID": "A",
            "supportedChannels": [ 1,  2 ],
            "typeName": "DB9",
            "typeNo": 7
         },
         "channel": 1,
         "frontPanel": "UA-3101-080",
         "status": "OK"
      },
      {
         "cable": {
            "BkNo": "",
            "ID": 0,
            "description": "No cable or it is not recognized, use on own risk",
            "lemoID": "",
            "supportedChannels": [ 1,  2],
            "typeName": "Unknown",
            "typeNo": 0
         },
         "channel": 2,
         "frontPanel": "UA-3101-080",
            "status": "OK"
        }
    ]
}
```

The table below describes the information returned by this command.

Parameter Name	Type	Description
channels	object	An array where each element is an object containing the scanning result for each channel. Each object has the parameters below.
cable	object	An object containing cable information with the parameters below
BkNo	string	The assigned number by Brüel & Kjær (for example, *“AO-0790”*)
ID	number	A unique ID used for each Brüel & Kjær cable type
description	string	Cable description
lemoID	string	Cable variant A character used to differentiate cables with the same Brüel & Kjær number.
supportedChannels	array	An array containing CAN channel’s number supporting the attached cable
typeName	string	CAN connector type name (for example, *“OBD-II”*)
typeNo	number	Cable type number
channel	number	CAN channel number
frontPanel	string	Front panel type
status	string	A string describing the status of the scanning (for example, *“OK”*)

2.4.9.4 /rest/rec/can/detectBaudRate

In addition to auto baud rate detection of CAN bus described in section 2.4.8, this command may be used to determine CAN-bus baud rate.

Please note: The auto baud rate detection requires CAN bus activity, otherwise it will fail.

URI:	http://<ip>/rest/rec/can/detectBaudRate
Supported Methods:	`PUT`
Valid States:	RecorderStreaming
Resulting State:	RecorderStreaming

Body:

```
{  "channels": [
          {
                 "channel": 1,
                 " baudRateTimeout ": 5,
                 "busType": "LowSpeed"
          },
          {
                 "channel": 2,
                 " baudRateTimeout ": 5,
                 "busType": "HighSpeedWithTerminator"
          }
   ]
}
```

The next table describes parameters used in the command body.

Parameter Name	Options	Type	Description
channels		object	An array where each element is an object which contains baud rate detection parameters for a channel defined in the parameters below
channel	*1, 2*	number	To select CAN channel

baudRateTimeout	*1 – 300*	number	Time out time in seconds
busType	*"LowSpeed", "HighSpeedWithoutTerminator", "HighSpeedWithTerminator"*	string	Selects the supported bus type

The information returned by this command looks similar to the following:

```
{  "channels": [
           {
                      "channel": 1,
                      "baudRate": 250000
           },
           {
                      "channel": 2,
                      "baudRate": 0
           }
   ]
}
```

2.4.10 /rest/rec/channels/all/disable

This command cancels a recording session.

URI:	http://<ip>/rest/rec/channels/all/disable
Supported Methods:	PUT
Valid States:	RecorderStreaming
Resulting State:	RecorderConfiguring
Body:	None

2.4.11 /rest/rec/onchange

This command is used to obtain status information while recording.

URI:	http://<ip>/rest/rec/onchange http://<ip>/rest/rec/onchange?last=<last update tag>
Supported Methods:	`GET`
Valid States:	All

Body:

```
{
   "moduleState": "Idle",
   "sdCardInserted": false,
   "buttonEnabled": true,
   "lastSdCardUpdateTag": 0,
   "transducerDetectionActive": false,
   "lastTransducerUpdateTag": 0,
   "canStartStreaming": false,
   "lastUpdateTag": 2,
   "recordingMode": "",
   "fanStatus": {
      "event": "",
      "speed": "",
      "mode": ""
   },
   "batteryStatus": {
      "event": ""
   },
   "temperatureStatus": {
      "event": "",
      "level": ""
   },
   "ptpStatus": "Locked",
   "inputStatus": "Unknown"
}
```

Much of the information may also be acquired using `GET /rest/rec/module/info` (see section 2.4.7).

The *update* tags indicate when certain changes have occurred. Whenever a change is detected, the ***lastUpdateTag*** field increases its value. If anything was changed in the state of the SD card, or if transducers were detected (TEDS detection), ***lastSdCardUpdateTag*** and/or ***lastTransducerUpdateTag*** will have changed values, indicating that some action may have to be taken to make sure the system is properly configured.

If "<last update tag>" is equal to ***lastUpdateTag***, the on-change request will wait for a change before returning. If no change has occurred, there is a time out of 30 seconds and the request will return with the same response as last request.

The client program must keep a copy of the last response to figure out what has changed.

moduleState	string	Current state of the module's state machine. See section 2.1.
sdCardInserted	boolean	Whether or not an SD card is inserted in the slot on the back of the module
buttonEnabled	boolean	Whether or not the button on the front of the module is configured to be used, for example, to cancel a recording
transducerDetectionActive	boolean	
lastSdCardUpdateTag	integer	See above
lastTransducerUpdateTag	integer	
lastUpdateTag	integer	
recordingMode	string	
fanStatus	object	Contains changes in fan speed and mode. It is relevant for frames only.
batteryStatus	object	Contains information on any changes in battery's predefined capacity levels. It is relevant for frames only.
temperatureStatus	object	Contains information on any changes in module's predefined temperature levels

ptpStatus	object	PTP state and is relevant if PTP is used for synchronization
canStatus	array of objects	

2.4.12 /rest/rec/destination/socket

When streaming to a single TCP socket (***destinations*** specified as *socket*), this command can be used to get the TCP port to connect to.

Refer to the protocol documentation in Chapter 5 for more information on how to use the streaming socket.

URI:	http://<ip>/rest/rec/destination/socket
Supported Methods:	`GET`
Valid States:	RecorderStreaming, RecorderRecording

Body:

```
{
        “tcpPort”: 1536
}
```

This body tells the client to connect to TCP port 1536 on the module to retrieve the streaming samples.

2.4.13 /rest/rec/destination/sockets

When streaming to multiple TCP sockets (***destinations*** specified as *multiSocket*), this command is used to get the TCP ports to connect to.

Refer to the protocol documentation in Chapter 5 for more information on how to use the streaming sockets.

URI:	http://<ip>/rest/rec/destination/sockets
Supported Methods:	`GET`
Valid States:	RecorderStreaming, RecorderRecording

Body:

```
{
        "tcpPorts": [ 1536, 1537, 1538, 1539 ]
}
```

This response tells that the client can connect to TCP ports 1536 – 1539 on the module to retrieve the streaming samples from each input channel. In this example, four input channels are enabled on the module, causing four TCP ports to be opened. Data from the first enabled input channel will be streamed to the first TCP port in the returned array, etc.

2.4.14 /rest/rec/measurements

Starts the streaming of samples to the destination (*socket*, *multiSocket* or *SD card*). Streaming starts as soon as the command is received and processed, and data needs to be consumed by the client before buffers overflow.

Sent to slaves first, then the master.

Please note: It is possible for a home-made application via Open API to control independent modules with different sample rates, even if they are in the same C- or D-frame. However, PULSE LabShop and BK Connect require the entire system to have a common sample rate.

URI:	http://<ip>/rest/rec/measurements
Supported Methods:	`POST`
Valid States:	RecorderStreaming
Resulting State:	RecorderRecording
Body:	None

2.4.15 /rest/rec/measurements/stop

Stops the streaming of samples to the destination (*socket* or *SD card*). Streaming is stopped as soon as the command is received and processed. If the destination is *socket/multiSocket*, this means that no effort is made to send any remaining packets for the current time (that is, channels may seem to end at different times).

URI:	http://<ip>/rest/rec/measurements/stop
Supported Methods:	`PUT`
Valid States:	RecorderRecording
Resulting State:	RecorderStreaming
Body:	None

2.4.16 Commands for PTP synchronization

When using two or more PTP (precision time protocol) synchronized modules, setting up requires some extra commands and parameters. The additions compared to the single-module setup are described here.

2.4.16.1 Master and Slaves

In a LAN-XI system, we operate with master and slaves regarding distributing the time and a trigger.

The time master, called the PTP Master, sends the time to the slaves.

The Trigger Master uses the time from the PTP system to tell the slaves when to fire the trigger.

Normally, the PTP Master and the Trigger Master are the same LAN-XI module. However, if the system includes an external time reference you can set ***preferredMaster*** to *false* in all LAN-XI modules and set the domain to the same as the time reference. Then you must define one of your LAN-XI modules as Trigger Master by setting ***triggerMaster*** to *true*.

A PTP Master module is always Trigger Master. A PTP slave module can be either Trigger Master or trigger slave.

A stand-alone LAN-XI module is automatically its own Trigger Master.

In a single LAN-XI frame, the module in slot 1 is automatically the Trigger Master.

In the above two examples, the ***triggerMaster*** is reported as *false* in the `syncmode` (`GET`) command, because the module is not sending triggers out on the Ethernet.

2.4.16.2 Domain

In LAN-XI, we operate with two domains: PTP and trigger. A domain is a number between 0 and 127.

Normally, the PTP and trigger domains use the same number, but in special cases they are different.

If you have a common time server on your network and two different LAN-XI systems on the same network, then select the same PTP domain for both LAN-XI system and the time server, and select different trigger domains for the individual LAN-XI systems.

2.4.16.3 TAI, UTC and Leap Seconds

- International Atomic Time (TAI) – see definition in Wikipedia®
- Coordinated Universal Time (UTC) – see definition in Wikipedia

See the latest news at:

- “leap seconds.pdf” from the International Bureau of Weights and Measures (BIPM) – https://www.bipm.org/utils/en/pdf/time_ann_rep/Time_annual_report_2019/9_Table2_TAR19.pdf
- “bulletinc.dat” from the International Earth Rotation and Reference Systems (IERS) – https://hpiers.obspm.fr/iers/bul/bulc/bulletinc.dat

On the wire, PTP uses TAI time. When delivering time to the user, UTC time is used. This requires information about the offset (leap seconds) between UTC and TAI; in 2019 this is 37 seconds.

BIPM sends out information six months ahead of adding an extra leap second. This information is distributed by the GPS. If your PTP Master has GPS receiver, you do not need to do anything.

Without GPS receiver, you can enter the new number on the module home page or use the Open API command `syncmode`.

When the offset changes, it is distributed to the PTP slaves and all modules store it in an EEPROM (electronically erasable programmable read-only memory).

Change of offset will not affect an ongoing measurement.

2.4.16.4 *States*

PTP synchronization adds several sub-steps to the initialization sequence – several of these require some time for systems to settle and synchronize between modules.

The module state has not been changed; no extra states need to be handled, but two new states need to be taken into account: PTP state and Input state.

The PTP state is used to tell when modules are settling and when they are locked, meaning that clocks are in sync.

The Input state similarly tells when the inputs are settled or synchronized.

2.4.16.5 *Flow*

Order	Action	REST Command	Resulting State		
			Module	PTP	Input
	Post-boot state		Idle	*Unknown*[*]	Unknown[*]
Master Slave(s)	Set synch-ronization mode	`/rec/syncmode`		*Unlock* *Settling* *Locking* *Locked*	
Slave(s) Master	Open Recorder	`/rec/open`	RecorderOpened		
Slave(s) Master	Create con-figuration	`/rec/create`	RecorderConfiguring		
Slave(s) Master	Configure input channels	`/rec/channels/input`			Settled
Slave(s) Master	Synchronize modules	`/rec/synchronize`			Synch-ronized
Slave(s) Master	Start internal streaming	`/rec/startstreaming`	RecorderStreaming		

	Get streaming port	`/rec/destination/socket` or `/rec/destination/sockets`			
Open Socket Connections (to all modules)					
Slave(s) Master	Start streaming	`/rec/measurements`	RecorderRecording		
Fetch Samples (from all modules)					
Master Slave(s)	Stop streaming	`/rec/measurements/stop`	RecorderStreaming		
Close Socket Connections (to all modules)					
Master Slave(s)	Finish recording session	`/rec/finish`	RecorderOpened		
Master Slave(s)	Close Recorder	`/rec/close`	Idle		

* PTP and input status may be *Unknown* if PTP, and the like, are not set up – but may also be any of the other states mentioned. This depends on the previous configuration and state. The example covers setting up PTP synchronization from scratch.

2.4.16.6 /rest/rec/syncmode

Sets PTP mode on the module. This command can also designate the preferred master in the PTP setup.

This command should generally be issued to the desired PTP Master first, then all slaves.

After this command is issued, the PTP state should change to *Locked* on all modules.

URI:	http://<ip>/rest/rec/syncmode
Supported Methods:	`GET, PUT`
Valid States:	Idle
Resulting State:	

Body:

```
{
        "synchronization": {
                "mode": "ptp",
                "domain": 45,
                "triggerDomain": 46,
                "preferredMaster": true
                “triggerMaster” : true
                “usegps”: false,
                “switchmethod”: “ptp”,
                "UtcTaiOffset": 37,
                “settime”: 1552478528000,
                “difftime”: 1500
        }
}
```

The synchronization object carries the PTP configuration.

mode	string	Set to *“ptp”* to enable PTP. Set to *“stand-alone”* to disable PTP.
domain	integer	The PTP domain to use for clock distribution. All time-synchronized modules must be set to the same PTP domain. No other systems should use this PTP domain, as this may severely impact the precision of the PTP, or may break synchronization altogether.
triggerDomain	integer	If this keyword is absent from the JSON object (the normal situation), the default value is the value from ***domain***. All trigger synchronized modules must have the same ***triggerDomain*** set.
preferredMaster	boolean	Set to *true* on the desired PTP Master, and *false* for the slaves. The PTP implementation will ensure that, at any given time, there is only one PTP Master for the PTP domain.

triggerMaster	boolean	If this keyword is absent from the JSON object (the normal situation), the default value is the value from ***preferredMaster***. When using an external PTP Master, like a time server, set ***preferredMaster*** to *false* on all modules and ***triggerMaster*** to *true* on one module in your system.
usegps	boolean	In frames with a GPS receiver, use this to timestamp measurements. Not used in PTP slave frames
switchmethod	string	This determines the type of LAN switch used: • *"ptp"*: A switch configured to use PTP • *"nonptp"*: A switch not using PTP (store-and-forward assumed) When mixing 100 Mb and 1 Gb LAN it is important to set this parameter. If wrong, you can get a phase error around 60° at 51200 Hz. This could, for example, happen when using a stand-alone module together with a frame. Default is *ptp* (used when this keyword is absent)
UtcTaiOffset	integer	Offset between UTC and TAI in seconds
settime	Int64	Milliseconds since midnight 1970-jan-01, including leap seconds Set the system time and PTP time on a master module. If applied (see ***difftime***), a PTP slave will receive new time from the master and will have to lock again. epochconverter.com has good information about time. *1552478528000* is 13 March 2019 12:02:08.000
difftime	Int64	Difference in milliseconds, default: *0* Use only on a master module. If the module system time differs less than ***difftime*** from ***settime***, then the module time is not altered.

If ***mode*** = *stand-alone*, there is no need to specify ***domain***, ***preferredMaster***, ***triggerMaster***, and ***usegps***; ***settime*** and ***difftime*** are optional.

If ***mode*** = *ptp*, you must specify ***domain***, ***preferredMaster***, and ***usegps***; ***triggerMaster***, ***settime*** and ***difftime*** are optional.

If you only want to change the time, then specify only ***settime*** and ***difftime***.

2.4.16.7 /rest/rec/synchronize

This command should be sent to all the PTP slaves before being issued to the master module. After this command is issued, the input state should change to *Synchronized* on all modules.

URI:	http://<ip>/rest/rec/synchronize
Supported Methods:	`PUT`
Valid States:	RecorderConfiguring
Resulting State:	
Body:	None

2.4.16.8 /rest/rec/startstreaming

Starts the internal streaming in the module. Send this command to all PTP slaves before the master module.

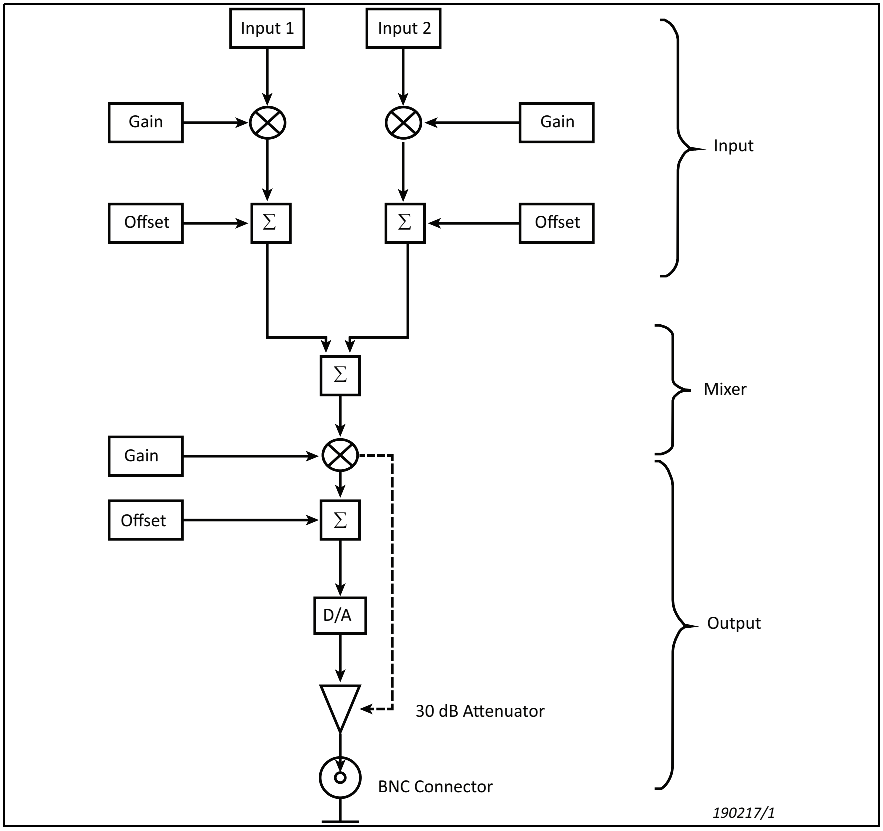
URI:	http://<ip>/rest/rec/startstreaming
Supported Methods:	`PUT`
Valid States:	RecorderConfiguring
Resulting State:	RecorderStreaming
Body:	None

Chapter 3

Output Generator

3.1 Model

Fig. 3.1 Generator model for one generator



The term 'input' in this model is a predefined waveform like a sine or a waveform streamed from the client to the generator module.

The mixing function can be summation, multiplication or pass. Pass is selected by the module when only one input is active.

The generator module Type 3160 has two outputs.

Please note: This means that the above model is duplicated.

It is recommended to check the output signal both in the time domain and the frequency domain. When using a new setting, it is easy to get a wrong signal. For example, gain at max then an offset added could give a clipped signal.

3.2 Command Sequence for Streamed Waveforms

RecorderOpen	
Generator/Prepare	*Start clock, reset mixer, enable output, etc.*
Generator/Output	*Setup Configuration*

Start sending data to generator 0 Start sending data to generator 1 When n*6000 samples sent to each generator	*n is minimum 3*
GeneratorStart	
....	*Binary Streaming ...*
Generator/Output 0	*Level change while streaming ...*
....	

Generator/Stop 0	
Generator/Stop 1	

RecorderClose	

3.3 Command Sequence for Built-in Waveforms

RecorderOpen	
Generator/Prepare	*Start clock, reset mixer, enable output, etc.*
Generator/Output	*Setup Configuration*

GeneratorStart	
….	
Generator/Output 0	*Level change*
….	

Generator/Stop 0	
Generator/Stop 1	

RecorderClose	

3.4 Commands

3.4.1 /rest/rec/generator/output

URI:	http://<ip>/rest/generator/output
Supported Methods:	`GET`, `PUT`
Valid States:	Idle
Resulting State:	

This is the Setup. Information is in JSON format, for example, from a `GET`:

```
{
   "outputs" : [
      {
               "number" : 1,
               “gain” : 0.75,
               “offset” : 0.0,
               "floating" : false,
               “inputs” : [
               {
                           “number” : 1,
                           “signalType” : “stream”,
                           “gain” : 1.0,
                           “offset” : 0.0,

                           “samplingRate” : 1
                           “source” : “socket”
                           “port” : 34709,
                           “errorcode” : 0
               },
               {
                           “number” : 2,
                           “signalType” : “none”
               }
               ]
      },
      {
               "number" : 2,
               “gain” : 1.00,
               “offset” : 0.0,
               "floating" : false,
               “mixfunction” : “sum”,
               “inputs” : [
               {
                           “number” : 1,
                           “signalType” : “sine”,
                           “gain” : 1.0,
                           “offset” : 0.0,
                        “frequency” : 40960.0
                        “phase” : 0.0,
                        “errorcode” : 0
                },
                {
                        “number” : 2,
                        “signalType” : “square”
                        “frequency” : 256.0
                        “gain” : 1.0,
                        “offset” : 0.0,
                }
                ]
        }
  ]
}
```

The properties are as follows:

number	integer	Output connector, or input source for the mixer
gain	float	Between *0.0* and *+0.999999* Values outside the range will be truncated. The signal is multiplied by the gain. Thus, it is only possible to attenuate the signal. If the output gain is below *0.0316227766*, the generator will increase the digital level 30 dB and turn on the 30 dB analogue attenuator to get a better signal-to-noise ratio on the analogue output.
offset	float	Values outside the range [*−0.999999*, *+0.999999*] will be truncated
floating	boolean	*true* or *false*

signalType	string	To each output you can assign 0, 1 or 2 inputs with different waveforms. If two inputs are chosen, a mixer is used, adding the two signals. Adjust the input gain and offset so that the output of the mixer does not exceed the range [–*0.999999*, +*0.99999*]. • *"stream"*: Means streaming from host to module, stream is only allowed in input number 1 • *"sine"*, *"linsweep"*, *"logsweep"*, *"random"*, *"p_random"*, *"dc"*, *"square"* • *"none"*: This input is not used Other waveforms may be included on request in the future
mixfunction	string	*"sum"*: Summation *"mul"*: Multiplication
errorcode	integer	*0* = success, other codes to be defined (relevant only in `GET` and `PUT` responses)

Unless otherwise noted, all input waveforms have a gain and an offset.

3.4.1.1 Stream Parameters

samplingrate	integer	The generator is always running at full speed. If your data to be streamed has a lower data rate, the generator can up-sample the data. Example with Type 3160 values: *0* = full speed 131072 samples per second *1* = 1/2 speed 65536 *2* = 1/4 speed 32768 *3* = 1/8 speed 16384
port	integer	The TCP port number where to connect (relevant only in `GET` and `PUT` responses)

Data streamed to the generator is 32-bit integers, with only the upper 24 bits used.

Each word is little-endian.

3.4.1.2 *Sine Parameters*

frequency	float	The frequency in Hz (for example, *1024.0*) Maximum depends on module type.
phase	float	Start phase *0.0 – 359.9999* degrees, default *0.0*

3.4.1.3 *Linsweep Parameters*

Linear sweep

start_frequency	float	The frequency in Hz (for example, *1024.0*) Maximum depends on module type.
stop_frequency	float	The frequency in Hz (for example, *10240.0*) Maximum depends on module type.
phase	float	Start phase, *0.0 – 359.9999* degrees
direction	integer	*0*: Frequency goes from start to stop, when stop is reached, frequency jumps to start and starts again. *1*: Frequency goes from start to stop, when stop is reached, frequency goes from stop to start.
hz_second	float	Sweep speed in Hz per second

3.4.1.4 *Logsweep Parameters*

Logarithmic sweep

start_frequency	float	The frequency in Hz (for example, *1024.0*) Maximum depends on module type.
stop_frequency	float	The frequency in Hz (for example, *10240.0*) Maximum depends on module type.
phase	float	Start phase, *0.0 – 359.9999* degrees

direction	integer	*0*: Frequency goes from start to stop, when stop is reached, frequency jumps to start and starts again. *1*: Frequency goes from start to stop, when stop is reached, frequency goes from stop to start.
decades_sec	float	Sweep speed in decades per second

3.4.1.5 Random Parameters

center_frequency	float	The frequency in Hz (for example, *1024.0*) Maximum depends on module type.
bandwidth	float	Bandwidth in Hz
hp_filter	boolean	Enable high-pass filter. Special high-pass filter used to remove the lower frequency part of random noise. F3dB = ***bandwidth***/100; Can only be activated if ***bandwidth*** ≤ max_bandwidth/8
pink_filter	boolean	Enable pink filter
gen_number	integer	Value *0 – 15* (up to 16 uncorrelated generator signals) ***gen_number*** controls the seed for the random generator

3.4.1.6 P_random Parameters

Pseudo random is a short time sequence with a configurable number of sine waves having random phase and equal amplitude.

The time signal is constructed using inverse FFT.

Center_frequency	float	The frequency in Hz (for example, *1024.0*) Maximum depends on module type.
bandwidth	float	Bandwidth in Hz
fftlines	integer	Legal values: *50*, *100*, *200*, *400*, *800*, *1600*, *3200*, *6400* Sequence length = fftlines × 2.56 [samples] Sequence length = (fftlines × 2.56)/ (bandwidth × 2.56) [sec]

nbseq	integer	Sequence repetition count Value = *1* or value ≥ *4*
pink_filter	boolean	Enable pink filter
gen_number	integer	Value *0 – 15* (up to 16 uncorrelated generator signals) ***gen_number*** controls the seed for the random generator, controlling the phase of the pseudo-random signal

3.4.1.7 DC Parameters

gain	float	The only parameter to “dc”, offset does not exist

3.4.1.8 Square Parameters

frequency	float	The frequency in Hz (for example, *1024.0*) Maximum depends on module type. Frequency above half of the maximum is not recommended.
phase	float	Start phase, *0.0 – 359.9999* degrees, default: *0.0*
repetitions	integer	Number of pulses, default: *0* = forever
dutycycle	float	Value *0.0 – 100.0, d*efault: *50.0*
highlevel	float	Should be in the range [*–0.999999*, *+0.999999*] Values outside the range will be truncated.
lowlevel	float	Should be in the range [*–0.999999*, *+0.999999*] Values outside the range will be truncated.
offset	float	Common offset added to high-and low-level, gain does not exist

3.4.2 rest/rec/generator/prepare

URI:	http://<ip>/rest/rec/generator/prepare
Supported Methods:	`PUT`
Valid States:	Idle
Resulting State:	

Body:

```
{
  "outputs" : [
    {
  "number" : 1
    },
    {
  "number" : 2
    }
  ]
}
```

This command will start the clock if not started already.

For the outputs mentioned in the JSON command, the signal generation will be stopped.

The input selector will be reset, so it is ready for a different waveform.

If the body looks like this:

```
{
  "outputs" : [
    {
  "number" : 2
    }
  ]
}
```

Only Output 2 will be reset. Output 1 will continue sending signal out (if already started).

3.4.3 rest/rec/generator/start

This command starts the generators in sync.

URI:	http://<ip>/rest/rec/generator/start
Supported Methods:	`PUT`
Valid States:	Idle
Resulting State:	

Body:

```
{
   "outputs" : [
      {
   "number" : 1
      },
      {
   "number" : 2
      }
   ]
}
```

To start all generators synchronized, we need a trigger. The module has a single trigger system used to start measurement or generators.

When using only a single module, the trigger is embedded in the `start` command. This maintains backward compatibility. In a system with more than one module, we need an `apply` command sent to the Trigger Master module, after all generators have received the `start` command. This rule applies even if there is only a single generator in the system.

In a single-module system, between a ***GeneratorSetup*** and a ***GeneratorStart***, it is not allowed to start a recording on a stand-alone module.

In a multi-module system, between a ***GeneratorSetup*** > ***GeneratorStart*** and the following `apply` command, it is not allowed to start a recording on a stand-alone module.

3.4.4 rest/rec/apply

URI:	http://<ip>/rest/rec/apply
Supported Methods:	`PUT`
Valid States:	Idle
Resulting State:	

Only used in a multi-module system. Multi-module systems can consist of a single frame (housing multiple modules), multiple frames (PTP) and/or several single modules (PTP). Send this command to the

Trigger Master module. This module will then send a trigger on the PTP system and/or the frame trigger bus. Now, all generators will start synchronized.

3.4.5 rest/rec/generator/stop

URI:	http://<ip>/rest/rec/generator/stop
Supported Methods:	`PUT`
Valid States:	Idle
Resulting State:	

Body:

```
{
   "outputs" : [
      {
   "number" : 1
      },
      {
   "number" : 2
      }
   ]
}
```

Chapter 4

General Commands

4.1 GPS

URI:	http://<ip>/rest/rec/gps
Supported Methods:	`GET`
Valid States:	All
Resulting State:	
Body:	None

Get data from the GPS receiver.

Only LAN-XI frames have a built-in GPS receiver. Only the frame controller module (slot 1) has access to the GPS receiver.

If the command is sent to a module without a GPS receiver or the GPS receiver is not ready, the following JSON data comes back:

```
{
   "gps" : {
      "quality" : "bad"
   }
}
```

If the GPS data is OK, the returned JSON data looks like this:

```
{
   "gps" : {
      "quality" : "ok",
      "latitude" : {
        "deg" : 55,
        "minute" : 48.99547958374023
        "char" : "N",
      },
      "longitude" : {
        "deg" : 12,
        "minute" : 32.02425003051758
        "char" : "E",
      },
      "utc_time" : {
        "year" : 2019
        "month" : 6,
        "day" : 19,
        "hour" : 13,
        "minute" : 56,
        "second" : 19.0,
      }
    }
}
```

4.2 Reboot

URI:	http://<ip>/rest/rec/reboot
Supported Methods:	`PUT`
Valid States:	All
Resulting State:	Idle
Body:	None

Unconditional reboot of the module.

NOTICE: If the module is recording, use the `stop` command described in section 2.4.15 before rebooting the module. Otherwise, some or all of the recorded data may be lost.

4.3 BatteryInfo

URI:	http://<ip>/rest/rec/batteryInfo
Supported Methods:	`GET`
Valid States:	All
Resulting State:	No state change
Body:	See below

This command is used to retrieve battery information. The command must be sent to the frame controller module that manages battery monitoring. Since the frame controller monitors batteries every 12 seconds, the returned data can be up to 12 seconds old.

Body:

```
{
   "supplySource": "battery",
   "remainingTime": 463,
   "batteries": [
           {
              "slot": 11,
              "capacity": 95,
              "status": " discharging",
              "current": -1164,
              "voltage": 16585,
              "serialNumber": 100713
            },         {
              "slot": 10,
              "capacity": 68,
              "status": " chargingSuspended",
              "current": -3,
              "voltage": 15468,
              "serialNumber": 100717
           }
        ]
}
```

The properties are as follows:

supplySource	string	If there is more than one supply source (for example, AC and battery), this field will show which source is selected to supply the frame. Supported supply sources are: *"AC"*, *"DC"* and *"battery"*
remainingTime	number	The remaining time in minutes. This information is available only if the ***supplySource*** is *battery*
batteries	object	An array where each element is an object containing battery-related information. Each object has the parameters below
slot	number	Frame slot number where the battery is inserted
capacity	number	Battery capacity in percentage, with the *0 – 100*% range
status	string	Battery charging status, which can be one of the following: *"charging"*, *"discharging"*, *"charged"*, *"chargingSuspended"*
current	number	Battery current in mA
voltage	number	Battery voltage in mV
serialNumber	number	Battery serial number

Any changes in supply sources (such as removing or inserting the battery or external power) is not reflected immediately. The update time depends on the frame type. The maximum time for Type C- and D-frames is 4 – 5 seconds while for the Type A-frame it is 20 seconds.

Chapter 5

Web-XI Streaming Protocol

The Web-XI® streaming protocol is used for transmitting samples through the network in the Recorder on the LAN-XI modules.

5.1 Time

Time in Web-XI is based on the concepts from the BK Connect time format. It is, however, expanded to support absolute time.

5.1.1 BK Connect Relative Time

The relative time in BK Connect is held in a 64-bit integer and contains a number of 'ticks'.

There are $2^{22} \times 3^2 \times 5^3 \times 7^2$ ticks per second, so each tick is approximately 4.33 picoseconds. The maximum duration of this time is approximately 461 days.

We can specify the following sample frequency families with this:

Name	Frequency	Frequency Range
65 kHz family	$2^n \times 3^0 \times 5^0 \times 7^0$	From 1 to 2^{22} Hz (4 MHz)
51.2 kHz family	$2^n \times 3^0 \times 5^2 \times 7^0$	From 25 to $2^{22} \times 25$ Hz (105 MHz)
256 kHz family	$2^n \times 3^0 \times 5^3 \times 7^0$	From 125 to $2^{22} \times 125$ Hz (524 MHz)
48 kHz family	$2^n \times 3^1 \times 5^2 \times 7^0$	From 375 to $2^{22} \times 375$ Hz (1.6 GHz)
44.1 kHz family	$2^n \times 3^2 \times 5^2 \times 7^2$	From 11025 to $2^{22} \times 11025$ Hz (46 GHz)

Here, a family is defined as a group of frequencies that 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 bits 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.

5.1.2 The Web-XI Absolute Time

The time for Web-XI is based on the ideas in the BK Connect time described above.

It is a 12-byte quantity in 2 parts:

The first part is the family: It is a 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

This family corresponds to a tick size of:
$2^{-k} \times 3^{-l} \times 5^{-m} \times 7^{-n}$ seconds

The second part is the count: It contains the number of ticks since 0 hours on 1 January 1970 (modified Julian Day: 40 587.0).

This is a 64-bit number.

5.1.3 Important Notes About the Time Format

5.1.3.1 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:

$2^k \times 3^l \times 5^m \times 7^n$ Hz

where:

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

5.1.3.2 PTP and LXI Compatibility

The PTP time-stamp format (as defined by IEEE 1588) is different from the format above:

Name	Data Type	Description
Seconds	6-byte	Time in seconds In LXI (an abbreviation for LAN extensions for instrumentation), this value is split into two parts: • The 4-byte ***Seconds*** field • The 2 byte ***Epoch*** field in the header
Nanoseconds	6-byte	In LXI, this is divided in the 32-bit ***Nanoseconds*** field and the 16-bit ***Fractional_Nanoseconds*** field

Zero time for this is 0 hours on 1 January 1970 (modified Julian Day: 40 587.0), which is the same as proposed for Web-XI format above.

The Web-XI application will have to convert to and from this format when implementing LXI and PTP specific protocols.

5.2 Data From the Device

The device generates data of different kinds:

- **Signal Data**: Measured data from analogue inputs
- **Interpretation Messages**: Describe how to interpret Signal Data
- **Aux Sequence Data**: Messages received on CAN inputs (LAN-XI Module Type 3058 only)
- **Data Quality**: Information about overload and other error conditions

Data is sent to the client as **Messages** in a TCP/IP stream.

The client does not need to enable or subscribe to messages – all messages are enabled by default.

Typically, the module will start off by sending Interpretation Messages for each enabled channel, followed by signal Signal Data and/or Aux Sequence Data messages for the duration of the recording.

Errors are reported using Data Quality messages that are added asynchronously to the stream.

5.2.1 Streaming

The module streams messages over a TCP/IP connection. The protocol will attempt to transfer all data and guarantees that any data loss caused by, for example, a slow network connection will be reported in a subsequent Data Quality message.

The device can either stream data from all input channels on a single TCP connection, or it can stream data from individual channels on their own, separate TCP connection. To select between single-socket or multi-socket operation, specify a ***destinations*** value of *socket* or *multiSocket* when sending the channel setup to `/rest/rec/channels/input` (see section 2.4.8.3).

5.2.2 Data Types and Value Domains

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

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

IDs (such as ***SignalId***) 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.

5.2.2.1 *Strings*

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

Name	Length (in bytes)	Contents
Count	2	The number of bytes in the UTF-8 string as an Int16 If ***Count*** is *0*, then the string is empty. ***Count*** may not be negative.
Bytes	Length	The actual content of the string ✎ **Please note:** The count is NOT the length of the string since a single character may be from *1* to *4* bytes in length.

5.2.2.2 *Streaming Message Format*

A message consists of a fixed-length header followed by content. The table below shows the message header.

Name	Length (in bytes)	Contents
Magic	2	The ASCII characters *BK*
HeaderLength	2	The length of the rest of the header up to but not including the message data length Currently, this is 20*.
MessageType	2	Identifies the content of the message.
Reserved1	2	For future use. Set to *0*.
Reserved2	4	Used for debugging.
Timestamp	12	Time of message
LengthOfMessageContent	4	Length of the message content in bytes

* If new fields are needed in the header, they should be appended after the ***Timestamp*** field. Existing fields must never be removed from the header. If these rules are followed, the header length will always increase for each new header version. The client can use the ***HeaderLength*** as a kind of header-version field.

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

MessageContent	LengthOfMessageContent	Depends on the message type.

The actual content depends on the message type. See the section on message types below.

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

5.2.3 Message Types

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

Message Type	Value	Description
Unknown	*0*	Value not set Should never be used in an actual message.
SignalData	*1*	Data values from a signal
DataQuality	*2*	Indicates the data quality of a certain signal's data. The quality message is typically only generated when the quality of a signal changes.
Interpretation	*8*	Describes how to interpret ***SignalData***.
AuxSequenceData	*11*	Similar to ***SignalData*** but contains relative time for non-equidistant data (for example, CAN data).

Each message type has an associated structure that is sent right after the header.

The following sections describe these structures.

5.2.3.1 *SignalData*

Messages of this type contain data values from a signal.

The message content is:

Name	Stream Type	REST Type	Description
NumberOfSignals	Int16	Number	Number of signals with data in this message.* The number of signals in the message should be non-zero.
Reserved	Int16		For future use Set to *0* when producing stream, ignore when consuming stream.

* It is up to the device to decide whether it wants to group signals into one message or not.

The following structure is repeated ***NumberOfSignals*** times:

Name	Stream Type	REST Type	Description
SignalId	Int16	Number	Identifies the signal that produced the following values
NumberOfValues	Int16	Number	Number of values The number of values should be non-zero.
Values	Array of values	Array of numbers	Data from the signal Number of values must be a multiple of the vector length.

The type of the values is given by the ***ContentDataType*** as defined in the Interpretation status message, see section 5.2.3.7.

5.2.3.2 *DataQuality*

Messages of this type contain quality information about a certain signal. The quality message is typically only generated when the quality of a signal changes.

The message content is:

Name	Stream Type	Description
NumberOfSignals	Int16	Number of signals with data in this message The number of signals in the message should be non-zero.

The following structure is repeated ***NumberOfSignals*** times:

Name	Stream Type	Description
SignalId	Int16	Identifies the signal that produced the quality.
Validity	DataValidity flags	Quality info
Reserved	Int16	For internal use Set to *4* when producing stream, ignore when consuming stream.

5.2.3.3 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 exists.

Name	Value	Description
Valid	*0*	Data is valid.
Clipped	*2*	The signal was clipped.
Invalid	*8*	The signal is invalid. Commands that return status information (for example, `GET /rest/rec/onchange?last=0`) give the status of the bits that are `OR`’ed into the ‘invalid’ bit: • *cf* = cable fault (includes CCLD overload) • *cmol* = common mode overload • *anol* = analogue overload The history within this run is also given: • *none* = no fault occurred during this record • *now* = the fault is there now • *prev* = the fault was there previously
Overrun	*16*	Overrun happened right before this value.

5.2.3.4 AuxSequenceData

Messages of this type contain non-equidistant data values from one or more signals. While ***SignalData*** messages are optimized for time-equidistant data, this message type contains a relative time, and thus is better suited for non-equidistant data.

Currently this message type is only used for CAN messages.

The message content is:

Name	Stream Type	REST Type	Description
NumberOfSignals	Int16	Number	Number of signals with data in this message* The number of signals in the message must be greater than zero.
Reserved	Int16		Reserved Set to *0*.

* It is up to the device to decide whether it wants to group signals into one message or multiple messages.

The following structure is then repeated ***NumberOfSignals*** times:

Name	Stream Type	REST Type	Description
SignalID	Int16	Number	Identifies the signal (where the data originates).
NumberOfValues*	Int16	Number	Number of values included in this message

* An ***AuxSequenceData*** message may be sent with ***NumberOfValues*** set at *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 inform that there will be no data at or before the given time (in BK Connect signal analysis parlance, this is called a ‘Synchronade’).

The following structure is then repeated ***NumberOfValues*** times:

Name	Stream Type	REST Type	Description
RelativeTime	Int32	Number	Time since the absolute time specified in the header
Value	A CAN message	Array of numbers	A single CAN message See *0* for the contents of this data message type.

5.2.3.5 *CAN Messages*

CAN messages are transmitted as ***AuxSequenceData*** described in the previous section. The value of ***AuxSequenceData*** is an array where each element is an object containing CAN data as shown in the table below.

Fields Name	Type	Description
status	byte	CAN bus and message status. Bit[0–1]: ControllerStatus (*0 = errorActive, 1 = errorPassive, 2 = BusOff*) Bit[2]: txErrorCounter (*0* = No error, *1* = txErrorCounter > 0) Bit[3]: rxErrorCounter (*0* = No error *1* = rxErrorCounter > 0) Bit[4]: LostMessage (*0* = OK, *1* = At least 1 message lost) Bit[5-7]: Reserved
CanMessageInfo	byte	This is a bitwise field, with the following meaning: Bit[0]: ExtendedID (*0* = 11-bit, *1* = *29-bit*) Bit[1]: RTR info (*0* = Send, *1* = RTR) Bit[2]: Data direction (*0* = Received, *1* = Transmitted) Bit[3–7]: Reserved

CanDataSize	byte	This is equal to the DLC (data length code) in the original received CAN message. Normally it is equal to number of bytes in the data field. However, it is not required to be the same for RTR messages.
Reserved	byte	Reserved for future use.
CanMessageID	4-byte	11-bit or 29-bit CAN message ID
CanData	8-byte	CAN data Number of valid data is defined by ***CanDataSize*** field.

The ***status*** field contains important information about the CAN-bus state and message errors. It can be compared to overload info for analogue data. The first two bits contain information about the CAN-bus state and are the most important bits (the names are defined by CAN protocol). *errorActive* means OK, while *errorPassive* means missing data (cable is removed/disconnected). *BusOff* is a fatal error and means that the channel is disabled, and all communication has been stopped. The channel can be enabled by reconfiguring it or by sending a `stop` and `start` command. The next two bits (rxErrorCounter and txErrorCounter) contain info whether an RX/TX error has occurred. The `onchange` command can be used to obtain the actual number of RX/TX errors. The last status field, LostMessage, is a sticky bit meaning that at least one message has been lost. This bit can be cleared by restarting the channel. The example below shows a complete Web-XI message containing 2 values that encapsulate CAN packages received from CAN channel 1.

Field Name	**Field Size**	**Value**	**Description**
Magic	2	*BK*	
HeaderLength	2	*20*	
MessageType	2	*11*	AuxSequenceData
Reserved1	2	*0*	
Reserved2	4	*0*	
TimeStamp	12	*45620142821*	Absolute time of arrival of first packet in this message

LengthOfMessageContent	4	*48*	2 + 2 + 2 + 2 + 2×20 = 48
NumberOfSequence	2	*1*	It means that all data is coming from same source.
Reserved	2	*0*	
SequenceID	2	*101*	CAN channel 1
NumberOfValues	2	*2*	2 CAN package
RelativeTime	4	*7345610*	
CanStatus	1	*0*	
CanMessageInfo	1	*0*	
CanDataSize	1	*3*	
Reserved	1	*0*	
CanMessageID	4	*0x7e0*	CAN frame ID (11 or 29 bits)
CanData	8	*5,6,7,0,0,0,0,0*	8-byte CAN data including padding
RelativeTime	4	*20242601*	
CanStatus	1	*0*	
CanMessageInfo	1	*0*	
CanDataSize	1	*3*	
Reserved	1	*0*	
CanMessageID	4	*0x7e0*	CAN frame ID (11 or 29 bits)
CanData	8	*5,6,7,0,0,0,0,0*	8-byte CAN data including padding

Dark shading = Web-XI header
Medium shading = ***AuxSequenceData*** header
Light shading = ***AuxSequenceData*** values that encapsulate CAN message

5.2.3.6 *Interpretation*

This message is sent when information about one or more signals changes. A piece of information about a signal, such as its unit, is called a descriptor.

All relevant descriptors are sent automatically when a new stream is opened.

The message contains one or more descriptors.

The content of a single descriptor is as follows:

Name	Stream Type	Description
SignalId	Int16	Identifies the signal to which this descriptor refers. If ***SignalId*** is *0* then the descriptor is for all signals.
DescriptorType	DescriptorType enum	Identifies the descriptor; see table of possible descriptor types below.
Reserved	Int16	Reserved for future use, set to *0*.
ValueLength	Int16	Length of value in bytes, not including any padding, that may have been added to ***Value*** to make it a multiple of a 32-bit word.
Value	Depends on descriptor type	The value of the descriptor This value must be a multiple of a 32-bit word.

5.2.3.7 *DescriptorType enum (Int16):*

The DescriptorType enum (Int16) contains:

Name	Value	Description	Type of Corresponding Descriptor Value	Default Value
DataType	*1*	Data type of a single value in signal	ContentDataType enum	None

ScaleFactor	*2*	Scale factor to multiply with each value in signal data to obtain a value in the specified unit.*	Float64	1.0
Offset	*3*	Offset to add to each value in signal data to obtain a value in the specified unit.	Float64	0.0
PeriodTime	*4*	Time between 2 consecutive values of the signal	Timestamp	None
Unit	*5*	The SI unit of the signal The corrected value (after applying scale factor and offset) will be in this unit.	See section 5.2.2.1	Empty string
VectorLength	*6*	Length of one value *0* means scalar	Int16	0
ChannelType	*7*	***ChannelType*** enum	Int16	1

* The scale factor is applied before the offset (CorrectedValue = ***ScaleFactor*** × ***signalValue*** + ***Offset***)

5.2.3.8 *ChannelType enum (Int16):*

The ChannelType (Int16) enum has the following possibilities:

Name	Value	Description
WebXiChannelType_None	*0*	Unknown type
WebXiChannelType_Input_Analog	*1*	Analogue input channel
WebXiChannelType_Input_Auxiliary	*2*	Auxiliary input channel
WebXiChannelType_CANBus	*3*	CAN bus
WebXiChannelType_Output_Analog	*20*	Analogue output channel
WebXiChannelType_Output_Auxiliary	*21*	Auxiliary output channel

5.2.3.9 *ContentDataType enum (Int16)*

The ContentDataType (Int16) enum has the following possibilities:

Name	Value	Description
Unknown	*0*	***ContentDataType*** not set (should never happen)
Byte	*1*	8-bit byte
Int16	*2*	16-bit integer
Int24	*3*	24-bit integer
Int32	*4*	32-bit integer
Int64	*5*	64-bit integer
Float32	*6*	32-bit float
Float64	*7*	64-bit float
Complex32	*8*	32-bit complex float
Complex64	*9*	64-bit complex float
String	*10*	UTF-8 string. Content is an Int16 length followed by the given number of bytes. See section 5.2.2.1

Chapter 6

General Concepts

This section briefly explains some of the basic concepts used throughout this document and the REST protocol.

6.1 REST

REST is an abbreviation for “representational state transfer”, meaning that changes are made to the settings on the device by transferring configurations rather than performing actions.

That is, if a switch is off and should be set on, the REST method would be to tell the device that the switch should be on, rather than flipping on the switch.

This concept is used throughout the REST protocol for any configuration task.

6.2 HTTP

The REST protocol uses the standard HTTP protocol for communication. The LAN-XI module includes a Web server that reacts to the REST commands and calls the relevant sub-systems.

A client may send an HTTP request to the server (LAN-XI module). This request contains:

- Header: Containing information about the request
- Body (optional): Carrying the contents of the request

When finished processing the request, the server sends an HTTP response to the client. This has the same basic header/body structure.

An HTTP request is made to a specific path, which determines the resource to target. Furthermore, the request specifies an HTTP method that tells what kind of request is made. The basic methods are as follows:

- `GET` – retrieve information from the requested path.

  The request body should be empty, and the information will be returned in the response body.
- `PUT` – update the requested resource.

  The new configuration should be sent in the request body. The response body should be empty.
- `POST` – create an element on the requested resource.

  The configuration should be sent in the request body. The response body should be empty.
- `DELETE` – delete an element on the requested resource.

  Both request and response bodies should be empty.

The request may look as follows (real examples picked up with Wireshark):

GET – request:

```
GET /rest/rec/onchange?last=0 HTTP/1.1
Host: 192.168.1.166
Connection: keep-alive
Accept-Encoding: gzip, deflate
User-Agent: Sonoscout/1.04.210 CFNetwork/672.0.8 Darwin/14.0.0
Accept-Language: da-dk
Accept: */*
```

GET – response:

```
HTTP/1.0 200 OK
Date: Tue, 19 Nov 2013 19:33:53 GMT
Connection: close
Server: Microsoft-WinCE/5.0
Content-Type: text/plain
Content-Length: 1330

{ "recordingStatus": { "measState": "Streaming", "errorState": 21, "recordingUri": "",
"timeElapsed": "00:01:17", "timeRemaining": "00:00:00", "spaceRemaining": 0,
"bufferStatus": { "cpu": { "fullPercentage": 0, "fullPercentageMax": 0 }, "dsp": {
"fullPercentage": 0, "fullPercentageMax": 0 } }, "channelStatus": [ { "rms":
0.000002026557922, "peak": 0.000038146972656, "cf": "none", "anol": "none",
"cmol": "none" }, { "rms": 0.000001430511475, "peak": 0.000008225440979, "cf":
"none", "anol": "none", "cmol": "none" }, { "rms": 0.000001430511475, "peak":
0.000006675720215, "cf": "none", "anol": "none", "cmol": "none" }, { "rms":
0.000001430511475, "peak": 0.000006437301636, "cf": "none", "anol": "none",
"cmol": "none" }, { "rms": 0.000001549720764, "peak": 0.000006914138794, "cf":
"none", "anol": "none", "cmol": "none" }, { "rms": 0.000001549720764, "peak":
0.000007033348083, "cf": "none", "anol": "none", "cmol": "none" } ] }, "moduleState":
"RecorderRecording", "sdCardInserted": false, "buttonEnabled": false,
"lastSdCardUpdateTag": 0, "transducerDetectionActive": false,
"lastTransducerUpdateTag": 2, "canStartStreaming": true, "lastUpdateTag": 409,
"recordingMode": "Single", "fanStatus": { "event": "", "speed": "", "mode": "" },
"batteryStatus": { "event": "" }, "temperatureStatus": { "event": "", "level": "" } }
```

PUT – request:

```
PUT /rest/rec/module/time HTTP/1.1
Host: 192.168.1.229
Accept-Encoding: gzip, deflate
Accept: */*
Content-Length: 13
Accept-Language: da-dk
Connection: keep-alive
Cache-Control: no-cache
User-Agent: Sonoscout/1.04.210 CFNetwork/609.1.4 Darwin/13.0.0

1383664328651
```

PUT – response:

```
HTTP/1.0 200 OK
Date: Tue, 05 Nov 2013 15:12:08 GMT
Connection: close
Server: Microsoft-WinCE/5.0
Content-Length: 0
```

6.2.1 Return Codes

Standard HTTP return codes are generally used to indicate the outcome of an operation. This description covers the ones typically encountered when working with the REST protocol.

6.2.1.1 Successful Operations

These codes indicate that the request was valid and has either been carried out or will be carried out later.

200 OK	The request succeeded. Any changes to the module state have already occurred by the time the response is returned. This means there is no need to `GET /rest/rec/onchange` after each request to check the module state. If you get **200 OK**, then the module has transitioned to the expected state and you can send the next request
202 Accepted	The request succeeded. Any changes are pending (may be sent to a PTP slave module, waiting for the master to trigger the execution, or may be handled asynchronously on the module)

6.2.1.2 Client Errors

In general, any 4*xx* return code points towards a client error, such as accessing a non-existing resource or providing invalid data.

400 Bad Request	There was a problem with the request. Bad JSON or invalid parameters will cause this. Attempting to configure output channels on a non-generator module will also result in **400**. The response body will include an error message explaining what the problem was. Typically caused by a bug in the client.
401 Unauthorized	The request was well-formed but the module is unable to carry it out, either because the current state of the module doesn't permit it, or the operation requires a license which is not available.
403 Forbidden	The request was well-formed but the module is unable to carry it out. Typically encountered when a request is not allowed in the state that the module is currently in, for example, attempting to start a recording without first sending a setup. The module will also return **403** if there is no SD card inserted and a request is made to start recording to it. Again, the response body will include an error message explaining why the request failed. Mainly caused by the client trying to do something that the current state does not allow.
404 Not Found	The resource does not exist. This should normally only be encountered if the client is trying to download a recording that has been deleted (or never existed). May otherwise be due to a bug in the client.
405 Method Not Allowed	Means the HTTP method used was not allowed on that resource. For example, `DELETE /rest/rec/channels/input` would result in a **405**. Typically caused by a bug in the client.

6.2.1.3 Server Errors

5*xx* errors are typically caused by errors on the server. They may be caused by the client's use of the server.

500 Internal Server Error	Something went wrong in the firmware, typically a failed API call or memory allocation. As with **400** and **403**, an error message is returned in the response body, though in some cases the content may only make sense to firmware developers
503 Service Unavailable	Caused by the client(s) having more than 10 concurrent connections to the module. As the module firmware does not support persistent connections, this error should only be encountered if the module stops responding to requests, that is, if the requests start 'hanging' or get stuck in the module

6.3 Data Formats

Different parts of the REST protocol use different data formats for exchanging information.

The formats used are primarily JSON and XML, but for certain uses simple plain text and BMP images have been used.

6.3.1 JSON

JSON is an abbreviation of 'JavaScript object notation' and provides a means of compact description of data structures and relations.

A detailed description is available at www.json.org.

Example:

```
{
    "a": 42,                    // Value for a has the integer value 42
    "b": "Hello, world!",       // Value for b is a string
    "c": [1, 2, 3, 4],          // c is an array of integers
    "d": {                      // d is an object
        "a": 5.1,               // ... with an element a which is a float
        "b": []                 // ... and an empty array b
    }
}
```

6.3.2 XML

Extensible Markup Language – XML – is another method of describing data structures.

More information can be found in this tutorial: www.w3schools.com/xml/.

Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<FrontPanel>
        <Version>0.0.0.1</Version>
        <Present>TRUE</Present>
        <Legal>TRUE</Legal>
        <Type>
              <Prefix>UA</Prefix>
              <Number>2107</Number>
              <Model/>
              <Variant>120</Variant>
        </Type>
        <Serialnumber>102</Serialnumber>
        <HwVersion>1.0.0.0</HwVersion>
        <DataBlocks/>
        <Extensions/>
</FrontPanel>
```

The example is taken from section 2.4.7.

A *FrontPanel* element surrounds the attributes *Version*, *Present*, *Legal*, etc.

Each element has a start tag and an end tag, for example <*Version*> and <*/Version*>. Empty elements can be written in shorthand as, for example, <*DataBlocks/*>, which acts as both start and end tag.

6.3.3 Other

In some places, other data types are used. These may include BMP images and simple clear text representations of single values.

6.4 State Machine

LAN-XI modules are governed by a central state machine. This is used to isolate certain uses from each other and makes it possible for clients to see what the module is currently doing.

That is, if a module is performing a firmware update, clients may discover that by looking at the module's state. While in the firmware update state, the module will not permit many other actions (that is, there are no valid state transitions), so opening the Recorder will not be possible at this point.

6.5 Time

The module operates with two types of time:

1) SystemTime, the time in the operating system
2) Measurement time or PTP time

Chapter 7

Calculating the Scale Factor

Sample data from the module is normalized in signed 24-bit two's complement. This is also known as Q23.

When you have a stream with normalized values, you need to convert the data to get the units you are interested in, for example, pascals. The conversion is described as:

$$calibratedValue = sampleValue \times scaleFactor$$

where *calibratedValue* is the value of interest, and *sampleValue* is the data received from the module. So, we need to calculate the *scaleFactor*:

$$scaleFactor = inputRange \times headroom / sensitivity$$

headroom is a factor used by the DSP (digital signal processor) to protect against signal clipping. It is normally 1.5 dB or 10^(1.5/20), which is about 1.18850.

An example:

If we are in 10 V range and the transducer has a sensitivity of 0.00918 V/Pa you get:

$$scaleFactor = 10\ V \times 1.18850 / (0.00918\ V / Pa) \cong 1295\ Pa$$

Thus, a floating-point value of 1 (also known as the full-scale value) will be 1295 Pa with the given microphone.

Appendix A

Firmware versions

The table contains an overview of LAN-XI firmware versions and their features.

Version	Date	Feature
2.10.0.344	January 4, 2021	Free, unlimited access to Open API
		Added parameters in `/rec/syncmode` (`GET`) in Open API
		CIC support in Open API
2.10.0.150	April 3, 2020	*T-insert* support for UA-3122 front panel in Open API
		Added *switchmethod* in `/rec/syncmode` in Open API
		Support for *T-insert ref* measurement with Type 3057
2.10.0.44	November 15, 2019	Fixes problem on Type 3057
2.10.0.27	October 3, 2019	New Open API features and improved long-term stability
		Last check-in in a series for TAI and UTC timing (UTC only before), GPS rollover and u-blox fixes
		CAN module can be used as master in a multi-module configuration, etc.

Version	Date	Feature
2.10.2.27 (cont.)	October 3, 2019 (cont.)	Support for battery information in Open API (JSON format) and FrameCtrl (XML)
		GPS-sync quality updated to settle in minimum 30 seconds
		Leap seconds handled when module is on
		Added *time* in `/rec/syncmode` (`GET`) in Open API
		Avoid unnecessary restart of PTP
		Added signal validity in streaming in Open API
		Added *triggerMaster*, *triggerDomain* in `/rec/syncmode` in Open API
2.9.0.552	August 12, 2019	Unlicensed Notar/SD recording; Notar no longer requires a license
		Added `/rec/gps` (`GET`)
		Fixed multi-module BK Connect Clock-ID error
		Retry CAN board initialization up to three times
		Corrected CIC generator level of Type 3160 module
		Front panel button disable support in Open API
		Implemented single channel TCP streams

Version	Date	Feature
2.9.0.458	May 9, 2019	Added fix for UA-310x-04x front panels
		Fixed generator DC calibration failure
		Corrected frame PTP slave internal configuration in Open API
		Added `/rec/reboot` (`PUT`)
		URI of commands case insensitive in Open API
		Added `rec/apply` (`PUT`) in Open API
		Fixed Dyn-X attenuators
		Added `/rec/syncmode` (`GET`) in Open API
		Support for built-in generator (not streaming) in Open API
		Fixed "Notar UI doesn't refresh in semi-continuous mode"
		CAN status/event support for BK Connect + CAN Self test
		"force standalone" sample clock generation for use in frame in Open API
		Added PCB ID for bridge completion board
		CAN support for BK Connect

Version	Date	Feature
2.9.0.458 (cont.)	May 9, 2019 (cont.)	Fixed bridge excitation voltage loop-back measurement (V-meas) bug
		Fixed bridge “activate” command bug
2.9.0.294	September 28, 2018	Added fix for Type 3056
		Support for UA-3122 front panel
		Fixed restoring factory firmware bug
		Support for high resolution generator level
		Fixed Notar for Type 3676 in Open API
		Notar handles incorrectly programmed TEDS 0.9 transducers
2.9.0.215	August 30, 2018	Released with BK Connect 2018.1
2.9.0.112	June 25, 2018	Released with BK Connect 2018.0
		Added REST command (`<ip>/rest/maxConnection`) to query the number of connections
		Fixed “disk full” error observed when recording files of size close to 4 GB
		Fixed thermometer initialization failure which prevented the temperature monitor to start

Version	Date	Feature
2.9.0.33	October 25, 2017	Fixed BKC file generation with 1.6 kHz bandwidth (small block/ record size) in Notar
		Fixed 24 to 32 conversion with small files in Notar
		Fixed decreasing firmware update progress percentage
		Improved response format `/rec/module/info` (`GET`)
		Fixed NULL pointer dereference on non-CAN modules for Notar/Open API
		Updated information on browser compatibility in Notar
2.7.0.568	May 17, 2016	Improved 30 V linearity and AUX timing
2.7.0.417	November 26, 2015	Released with PULSE 20.0.0.455
		SNTP service implemented for LAN-XI
		Support for set nulling on bridge module, including REST
		Support for LAN-XI Bridge Input Module Type 3057 in Open API
		ConnectorSelect and *bridgeNulling* dependent on module type in Open API

Version	Date	Feature
2.7.0.417 (cont.)	November 26, 2015 (cont.)	Changed *bridgeCompletion* from *Full* to *None*
		Support for front panels: UA-3100 and UA-3102
		Bridge fast nulling support which cases on analogue hardware version 3.10 or greater
2.7.0.185	June 9, 2015	Released with PULSE 19.0
		Fixed TEDS bugs
		Support for 1-wire bus short detection
		Changed GPS system to handle u-blox GPS chip
		UDP streaming, added Notar UI to specify destination IP address and port
		Clock synchronization system changed, full support for GPS
		Fixed repeating of streaming output for duration of ~32 seconds
		Fixed ramping and sweeping of sine output
		Added A-Frame with GPS in frame_tab.h
		Added GPS support from REST interface to PTP
		Added frame support for multi-module streaming in Open API

Version	Date	Feature
2.7.0.185 (cont.)	June 9, 2015 (cont.)	Timestamp in streaming headers changed to 64-bit PTP format, 32-bit sec + 32-bit subseconds
		Added ***time*** parameter to REST command `syncmode` so it is possible to set the PTP time

Please note: Support for new modules (for example, the CAN module) implemented before they were introduced is not included in the table above. The same goes for various clean-ups and internal test-code as well as support for new FPGAs (field-programmable gate array), displays, etc.

Some features are bundled on newest date.

Index

BE 1872 – 15

www.bksv.com