LAN-XI Open API Developer Package
BZ-5959-D

Reference Guide – version 17

For more information on LAN-XI data acquisition hardware, please see our website at www.bksv.com/lanxi and github.com/hbk-world

Created and maintained by HBK Instrumentation Group

September 2020 BE 1872 – 11

1 Contents

2 Document History

Version	Author	Date	Changes
1	MREZAI	2012-03-29	Initial version with full command summary
2	PBC	2014-01-30	Master-Slave document with tables and generalization
3	KELK	2014-04-30	Specific “Open API” limited version and wireshark samples
4	EWALTON	2014-06-11	Proofed
5	KELK	2015-06-02	Updated for preliminary support of 3057 Bridge Module
6	KELK	2015-07-02	Release For Use with 3057 Bridge Module
7	KELK	2016-04-18	Generator Output added
8	KELK	2016-11-03	Added “fileFormat” as described by CHA
9	KELK	2018-04-09	Generator updated
10	LET	2019-06-20	Added all generator wave forms Added apply command Added reboot command Added GPS command
11	MREZAI	2019-06-26	Added CAN related description
12	MREZAI	2019-08-05	Added support to retrieve battery information
13	CHANSEN	2019-09-24	Corrected AuxSequenceData description
13	CHANSEN	2019-09-24	Removed Packages message type, this was never supported by LAN-XI firmware
13	CHANSEN	2019-09-24	Corrected AuxSequenceData description
14	LET MREZAI	2020-03-02 2020-03-13	Added switchmethod to syncmode Added T-insert support for UA-3122 front panel
15	KELK	2020-03-30	Supported modules and faceplates described
16	MREZAI	2020-06-24	Added CIC support
17	KELK	2020-09-22	Data Validity better described. github reference included. Copyright changed to HBK

3 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”.

In order to use the protocols with LAN-XI hardware, every LAN-XI module used must have a LAN-XI API License BZ-5959-L-N01 installed. This is linked to type- and serial-number.

4 Supported Commands, Modules and Front panels

Some commands are only supported for specific modules – e.g. commands for bridge settings are only relevant for the 3057 Bridge module. The 3676 and 3677 LAN-XI *Light* modules only work one at a time – hence the PTP setup is irrelevant on these modules. The only LAN-XI module not supported at all is the 3056 Tacho/Aux module.

There are numerous front panels – aka detachable fronts – for LAN-XI. These work like usual. This means that a faceplate which normally only works with e.g. the 3057 Bridge module on BK Connect also only works with this module when using the Open API.

5 Recorder

For historic reasons, the Open API is built on top of the ‘Notar’ application – also known as “the Recorder”. This has defined the various states that the application can – and must – go through.

The Recorder application 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.

5.1 State machine

The Recorder application operates using several state domains. The main states are the module states. The module states relevant to the Recorder application are illustrated in Figure 1.

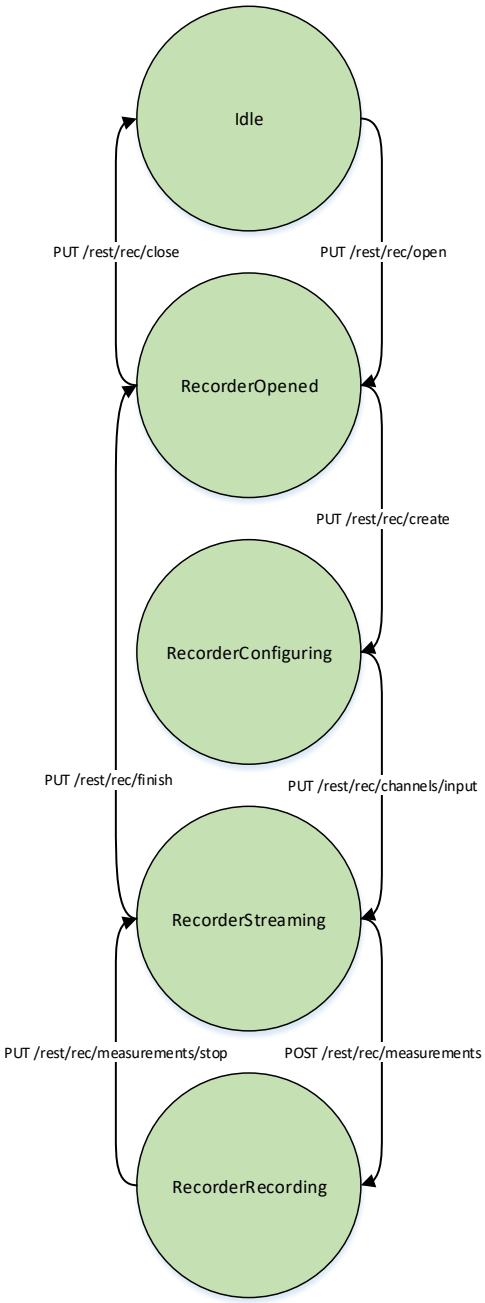

Figure 1 - Module states relevant to the Recorder application – and transitions between the states

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

5.2 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.

5.3 Input Related Commands

5.3.1 /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*.

5.3.2 /rest/rec/close

This command closes the Recorder application on the module.

URI	http://<ip>/rest/rec/close
Supported methods	PUT
Valid states	RecorderOpened
Resulting state	Idle
Body	None

5.3.3 /rest/rec/create

This command creates a recording configuration for the Recorder application.

URI	http://<ip>/rest/rec/create
Supported methods	PUT
Valid states	RecorderOpened
Resulting state	RecorderConfiguring
Body	None

5.3.4 /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) , incl. 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

5.3.5 /rest/rec/module/info

This command returns information about the module.

URI	http://<ip>/rest/rec/module/info
Supported methods	GET
Valid states	All

Body example from a 3057 module (JSON):

Note that, 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 5.3.6.3 Data Exchange. 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 5.1.
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 ints	Sample rates supported by the module, presented in an array.
syncModeChangeSupported	boolean	

Table below shows some other properties support by CAN module and not shown in the above example.

supportedHatsChannelPairs	Array of array	An array where each element is an array containing a pair of channels supporting HATS (AES) input (e.g. *[[3,7], [4,8]]*). This property is not shown in the above example and currently only supported by CAN module.
numberOfChannels	Integer	Number of CAN channels supported by the module.
supportedLowSpeedChannels	array of integer	An array of integer containing channel-number supporting Low-speed CAN bus type.
supportedHighSpeedChannels	array of integer	An array of integer containing channel-number supporting High-speed CAN bus type.
supportedObd2Channels	array of integer	An array of integer containing channel-number supporting OBD-II.
supportedLowSpeedBaudrates	object	Supported baud rate range for LowSpeed BUS type. (E.g. *{"auto": 0, "min": 10000, "max": 125000 }*)
supportedHighSpeedBaudrates	object	Supported baud rate range for HighSpeed BUS type. (E.g. *{"auto": 0, "min": 10000, "max": 1000000 }*)
supportedModes	array of strings	An array of string describing supported modes. (E.g. *["Passive", "Active", "Write"]*)
supportedBusTypes	array of strings	An array of string describing supported BUS type. (E.g. *["LowSpeed", "HighSpeedWithoutTermainator", "HighSpeedWithTerminator"]*)

5.3.6 Input channel configuration

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

5.3.6.1 *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 (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:

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*}
channels	object	An array of channel and transducer parameters, each element being an object consisting of the following parameters:
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. 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 supply 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 supply is "DC Current"
bridgeRemoteSense Wiring	string	*{None, SinglePos, Double}* None -> No compensation for voltage drop over cables Single → Compensation calculated by module at meas start Double → Analog HW compensation
bridgeShunt	boolean	*{true, false}* Tells 3057 that 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.
hats	boolean	Enables the use of AES/HATS (Head And Torso Simulator) inputs. This is supported by CAN modules - requiring two input channels (called pair-channels). Pair-channel info can be acquired with the /rest/rec/module/info command.
channel	integer	The channel number (1 is the first channel).
connectorSelect	string	*{BNC, LEMO, Charge, Bridge, Diff Charge, T-insert, T-insert ref}* See note below this table.
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, 1Vpeak, 10Vpeak, 31.6 Vpeak}*
transducer	object	Transducer setup.
fileFormat	string	Specifies the format of the recording file. Options are • "wav" to produce a recording in WAV format, or • "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 WAV format.

Note: The two *'T-insert' and 'T-insert ref'* options of the 'connectorSelect' parameter are a special variant of charge input 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 3122--030 front.

The next table shows parameters specific to the CAN module (type 3058). These are ignored by non-CAN modules and may be omitted.

canChannels	object	An array of CAN channel parameters, each element being an object consisting of the following parameters:
channel	integer	CAN channel number.
name	string	A descriptive name for the CAN channel defined by user
enabled	boolean	Enables (true) or disables CAN channel
destination	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 zero for automatic baud rate detection.
baudRateDetectionTimeout	integer	Automatic baud rate detection timeout 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.

Note: Run 5.3.5 /rest/rec/module/info to see complete info, layout and exact value ranges for a given module.

5.3.6.2 /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 5.3.6.1

5.3.6.3 */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	RecorderConfiguring
Resulting state	(PUT) RecorderStreaming
Body	See 5.3.6.1

5.3.6.4 */rest/rec/channels/cic*

This command prepares the desired input channels for CIC measurement and must be sent prior to */rest/rec/channels/input* command. When issuing the input configuration setup (after CIC setup), then the selected channels must be enabled and the connectorSelect parameter is set to LEMO. 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 number	Array of selected input channel number 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 from 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 degree

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 generator simultaneously. Therefore, using a generator module like 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.

5.3.6.5 */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 5.3.6.1

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 being an object consisting of:
channel	integer	*{1..3}*
nulling	string	*{None, Automatic, Reset}*

5.3.6.6 */rest/rec/can/obd2*

This command is used to request an OBD-II message and supported by CAN module (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 or delete to or from the internal list. Table below describes OBD-II parameters.

Parameter name	Options	Type	Description
action	KeepList, Overwirte, DeleteAll	String	Selects the action to perform. • **KeepList**: Keeps the internal list and performs an add or delete operation on it. This is 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 description below.
Obd2Messages		Object	An array of OBD2 messages where, each element being an object consisting of the following parameters:
channel	1, 2	Number	To select CAN channel
messageID	11 or 29 bits ID	Number	CAN message ID
cycleTime	1 - 65535 ms	Number	Time in millisecond 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=11bit 1=29bit) Bit[1]: RTR (1=Remote Transmission Request)

			Bit[2-7]: Reserved Note: When RTR is set then no data are allowed.
dataSize	0-8	Number	Number of data bytes in the next field.
data	Up to 8 bytes of data	Array of Number	OBD-II data to be send 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 combination like *Overwrite* action with DELETE method type is not allowed. See table above for supported methods with the selected action parameter.

Note: If the *mode* parameter described in section 5.3.6.1 is not set to "Write" mode, then the OBD-II messages will not be transmitted to CAN bus.

5.3.6.7 /rest/rec/can/sendMessages

This command is used to transmit a number of CAN messages once. The *mode* parameter described in section 5.3.6.1 must be set to "Write" mode in order to use this command.

URI	http://<ip>/rest/rec/can/sendMessages
Supported methods	
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 consisting of the following parameters:
channel	1, 2	Number	To select CAN channel
messageInfo	See description	Number	This is a bitwise field, with the following meaning Bit[0]: ExtendedID (0=11bit 1=29bit)

			Bit[1]: RTR (1=Remote Transmission Request) Bit[2-7]: Reserved Note: • RTR is used to request CAN message from a specific CAN device.
dataSize	0-8	Number	Number of bytes in the data field.
messageID	11 or 29 bits ID	Number	CAN message ID
data	Up to 8 bytes of data	Array of Number	Data to be send to CAN bus. Max 8 bytes

5.3.7 Detect commands

5.3.7.1 /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.

5.3.7.2 /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

5.3.7.3 /rest/rec/can/detectCables

This command is used to detect and gather information on attached cables 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 following parameters.
channel	Number	CAN channel number.
status	String	A string describing the status of the scanning (e.g. “OK”).
frontPanel	String	Front panel type.
cable	Object	An object containing cable information with the following parameters.
ID	Number	A unique ID used for each B&K cable type
BkNo	String	The assigned number by B&K (e.g. “AO-0790”)
typeName	String	CAN connector type name (e.g. “OBD-II”)
typeNo	Number	Cable type number.
lemoID	String	Cable variant. A character used to differentiate cables with the same BkNo.
supportedChannels	Array	An array containing CAN channels number supporting the attached cable
description	String	Cable description

5.3.7.4 /rest/rec/can/detectBaudRate

In addition to auto baud rate detection of CAN bus described in section 5.3.6, this command may be used to determine CAN bus baud rate. Please note that 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 containing baud rate detection parameters for a channel defined in the following.
Channel	1, 2	Number	To select CAN channel

baudRateTimeout	1-300	Number	Timeout time in second.
busType	LowSpeed, HighSpeedWithoutTerminator, HighSpeedWithTerminator	String	Selects the supported bus types

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

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

5.3.8 /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

5.3.9 /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 5.3.5 *GET /rest/rec/module/info*

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 with ”lastUpdateTag”: the onchange request will wait for a change before returning, if no change occurred there is a timeout of 30 seconds and the request will return with the same response as last request.
The client program must keep a copy of last response to figure out what has changed.

moduleState	string	Current state of the module’s state machine. See section 5.1.
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 to, for example, cancel a recording.
transducerDetectionActive	boolean	
lastSdCardUpdateTag	integer	See above
lastTransducerUpdateTag	integer	
lastUpdateTag	integer	
recordingMode	string	
fanStatus	object	Contains changes in fan speed and mode. Relevant for frames only.
batteryStatus	object	Contains information on any changes in battery’s predefined capacity levels. It is relevant for frame only.
temperatureStatus	object	Contains information on any changes in module’s predefined temperature levels.
ptpStatus	object	PTP state and is relevant it PTP is used for synchronization
canStatus	Array of object	

5.3.10 /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 section 8 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.

5.3.11 /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 section 8 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.

5.3.12 /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.

URI	http://<ip>/rest/rec/measurements
Supported methods	POST
Valid states	RecorderStreaming
Resulting state	RecorderRecording
Body	None

5.3.13 /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

5.3.14 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.

5.3.14.1 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 standalone 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.

5.3.14.2 Domain

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

5.3.14.3 TAI, UTC and leap seconds

International Atomic Time (TAI, from the French name *temps atomique international*).
Coordinated Universal Time (UTC).
See leap seconds.pdf from “International Bureau of Weights and Measures (BIPM)” or bulletinc.dat from “INTERNATIONAL EARTH ROTATION AND REFERENCE SYSTEMS SERVICE” (IERS).

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 6 months ahead of adding an extra leap second. This information is distributed by the GPS.
If your PTP master has GPS-receiver, you don’t 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.
Change of offset will not affect an ongoing measurement.

5.3.14.4 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.

5.3.14.5 Flow

Order	Action	REST command	Resulting module state	Resulting PTP state	Resulting input state
	Post-boot state		Idle	Unknown*	Unknown*
Master Slave(s)	Set synchronization mode	/rec/syncmode		Unlock - Settling - Locking - Locked	
Slave(s) Master	Open Recorder	/rec/open	RecorderOpened		
Slave(s) Master	Create configuration	/rec/create	RecorderConfiguring		
Slave(s) Master	Configure input channels	/rec/channels/input			Settled
Slave(s) Master	Synchronize modules	/rec/synchronize			Synchronized
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, etc. 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.

5.3.14.6 /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 same PTP domain. No other systems should use this PTP domain, as this may severely impact the precision of 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 frame with GPS-receiver, use this to timestamp measurements. Not used in PTP slave frames. Default is false.
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 1Gb LAN it is important to set this parameter. If wrong you can get a phase error around 60 degrees at 51200 Hz. This could e.g. happen when using a standalone 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	Milli-seconds since midnight 1970-jan-01, incl. leap seconds. Set the systemtime 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. https://www.epochconverter.com has good information about time. 1552478528000 is 13. March 2019 12:02:08.000
difftime	Int64	Difference in milli-seconds. Default 0. Use only on a master module. If the module system time differ 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”

5.3.14.7 /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

5.3.14.8 /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

6 Output Generator

6.1 Model

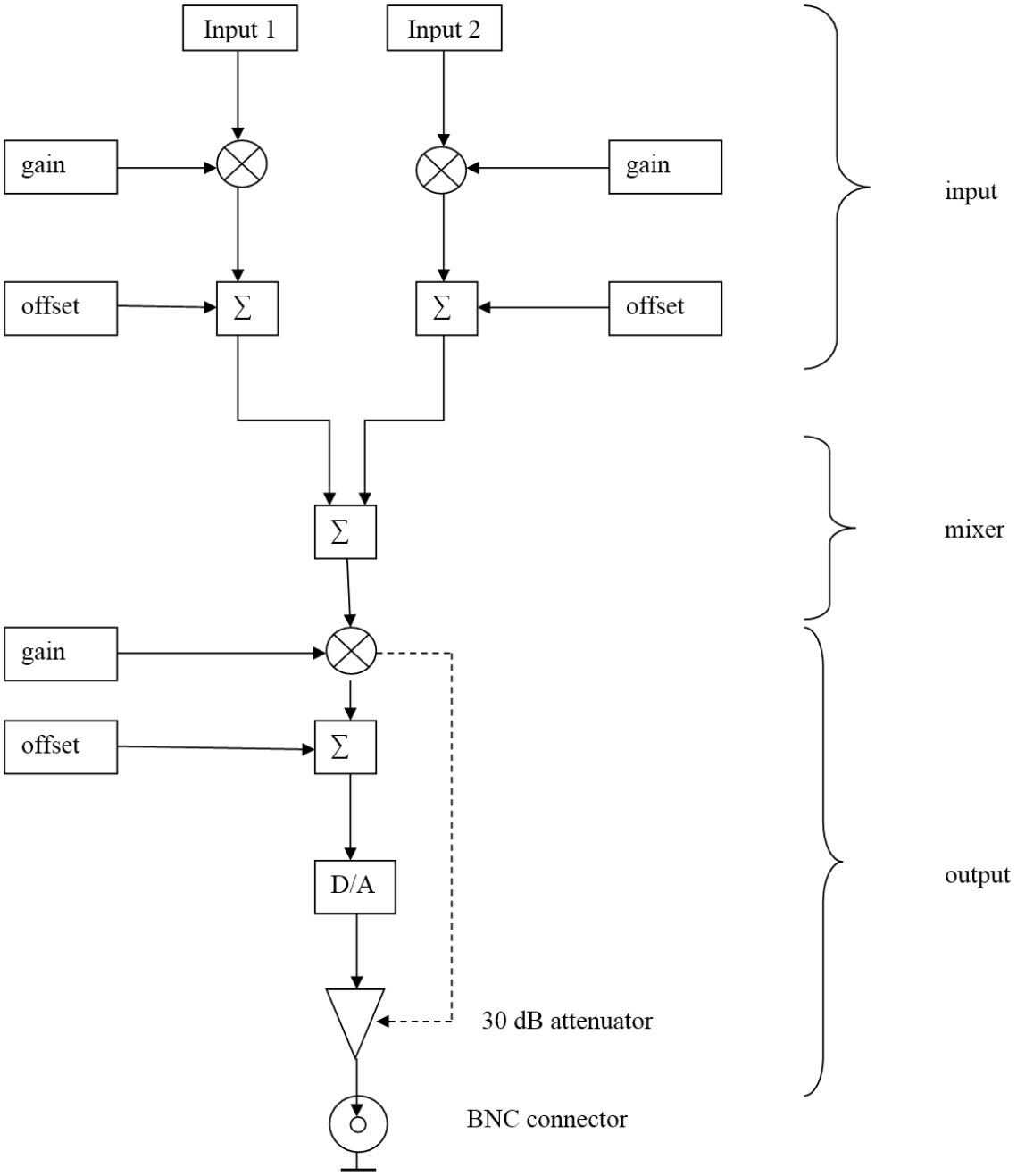

Figure 2. Generator Model for ONE Generator

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 (Selected by the module, when only one input is active).

The generator module 3160 has two outputs. **This means that the above model is duplicated**.

It is recommended to check the output signal both in time domain and frequency domain, when using a new setting, it is easy to get a wrong signal. Ex: gain at max then add an offset. This could give a clipped signal.

6.2 Command Sequence for streamed waveform

- 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

6.3 Command Sequence built in waveform

- 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

6.4 Commands

6.4.1 /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, e.g.: 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 *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 30dB and turn on the 30dB analog attenuator to get a better signal/noise ratio on the analog 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” meaning 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 **Response**)

Unless otherwise noted all input waveforms has a gain and offset.

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 3160 values: 0=full speed 131072 samples / 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 **Response**)

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

Each word is in little endian.

Sine parameters

frequency	float	The frequency in Hz. Ex: 1024.0, max is module type dependent.
phase	float	Start phase 0.0 – 359.9999 degrees, default 0.0

Linsweep parameters

Linear sweep

start_frequency	float	The frequency in Hz. Ex: 1024.0, max is module type dependent.

stop_frequency	float	The frequency in Hz. Ex: 10240.0, max is module type dependent.
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.

Logsweep parameters

Logarithmic sweep

start_frequency	float	The frequency in Hz. Ex: 1024.0, max is module type dependent.
stop_frequency	float	The frequency in Hz. Ex: 10240.0, max is module type dependent.
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.

Random parameters

center_frequency	float	The frequency in Hz. Ex: 1024.0, max is module type dependent.
bandwidth	float	Bandwidth in Hz.
hp_filter	boolean	Enable high pass filter. Special highpass 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.

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. Ex: 1024.0, max is module type dependent.
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.

Dc parameters

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

Square parameters

frequency	float	The frequency in Hz. Ex: 1024.0, max is module type dependent. Frequency above half of the maximum is not recommended.
phase	float	Start phase 0.0 – 359.9999 degrees, default 0.0
repetitions	Integer	0 = forever. (default) Number of pulses
dutycycle	float	Value range 0.0 to 100.0. Default 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 lowlevel, gain don’t exist

6.4.2 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 startet)

6.4.3 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 only has a single trigger system, used to start measurement or start 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 triggerMaster module, after all generators has 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 standalone module.
In a multi module: between a GeneratorSetup -> GeneratorStart and the following 'apply', it is not allowed to start a recording on a standalone module.

6.4.4 rest/rec/apply

URI	http://<ip>/rest/rec/apply
Supported methods	PUT
Valid states	Idle
Resulting state	

Only used in a multi module system = single-frame, multi-frame(PTP) and/or several single modules(PTP). Send this command to the triggerMaster module. This module will then send a trigger on the PTP system or/and the frame trigger bus. Now, all generators will start synchronized.

6.4.5 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
    }
   ]
}
```

7 General commands

7.1 gps

URI	http://<ip>/rest/rec/gps
Supported methods	GET
Valid states	All
Resulting state	
Body	None

Get data from the GPS-receiver.
Only frames has a GPS-receiver. Only 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 GPS-data is ok, the return JSON data looks like:

```
{
  "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,      }
  }
}
```

7.2 Reboot

URI	http://<ip>/rest/rec/reboot
Supported methods	PUT
Valid states	All
Resulting state	Idle
Body	None

Unconditionally reboot of the module. Be careful - valuable measurements may be lost.
Sleep approx. 30 seconds after this command.

7.3 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 is responsible and 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 (e.g. 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 supply source is battery.
batteries	Object	An array where, each element is an object containing battery related information. Each object has the following parameters.

slot	Number	Frame slot number where 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/inserting battery or external power) is not reflected immediately. The update time depends on the frame type. The maximum time for the C- and D-frames is 4-5 seconds while for the A-frame is 20 seconds.

8 Web-XI Streaming protocol

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

8.1 Time

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

8.1.1 BK Connect Relative time

The relative time in PULSE Reflex is held in a 64-bit integer and contains a number of "ticks".

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

We can specify the following sample frequency families with this:

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

Here a family is defined as a group of frequencies 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.

8.1.2 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:

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

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.

8.1.3 Important notes about the time format

8.1.3.1 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:

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

where

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

8.1.3.2 PTP and LXI Compatibility

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

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

Zero time for this is 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.

8.2 Data from the device

The device generates data of different kinds:

- **Results** are, for instance, measured data or the result of some signal analysis done on the device. Results come from a "Signal" in the object model.
- **Data Quality** is also a kind of result but is classified separately here because it is handled a little different. (Value changes are transmitted instead of all values.)
- **Events** describe something that has happened on the device, such as a state change, a trigger or a configuration change.

Such data is sent to the client as **Messages** and can be sent with one of the following methods:

- **Streaming:** The Client can request that certain message types are sent in a TCP/IP stream**.** A stream can contain all message types. The protocol for doing this is described in this document
- **Events:** The client can request information about overloads, etc., from a module. This is done with the REST protocol.

This section first describes the different methods for sending data to the client. It goes on to describe the different types of message that can be sent.

8.2.1 Streaming

The module can deliver results, quality data and events over a TCP/IP connection. This is called streaming. The protocol tries to transfer all requested data and guarantees that any data loss caused e.g. by a slow network connection will be reported as a status event.

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 5.3.6.3).

8.2.2 Timing of messages in the stream

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

For instance, the client will need to

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

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

- Data describing an input channel's data must be transmitted before the data they describe (for example, quality of channel 1 before channel 1.)
- General status data must be sent before any of the input data.

The problem is that the client needs to know when it is ok to discard old data.

To facilitate this, the device should emit a "heartbeat" in the data stream at certain intervals.

The device guarantees that no more data will be sent with a timestamp before the one given in the heartbeat message.

8.2.3 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 in fact is 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.

8.2.3.1 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 UTF8 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. (Note that the count is NOT the length of the string since a single character may be from 1 to 4 bytes in length.)

8.2.3.2 *Streaming message format*

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

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[1].
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

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.

8.2.4 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 signals data. The quality message is typically only generated when the quality of a signal changes.
Interpretation	8	Describes how to interpret Signal Data.
AuxSequenceData	11	Similar to signal data but contains relative time for non-equidistance data (e.g. CAN data).

[1] 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 “Header Length” as a kind of Header version field.

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

The following sections describe these structures.

8.2.4.1 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.[2] 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.

The following structure is repeated "NumberOfSignal" 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 Number	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 8.2.4.5.

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

8.2.4.2 *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 "NumberOfSignal" 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.

8.2.4.3 *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 exist.

Name	Value	Description
Valid	0	Data is valid
Clipped	2	The signal was clipped
Invalid	8	The signal is invalid. Commands that return status – e.g. *GET /rest/rec/onchange?last=0* – gives the status of the bits that are OR'ed into the "Invalid" bit: • "cf" = Cable Fault – incl. CCLD Overload • "cmol" = Common Mode Overload • "anol" = Analog Overload The history within "this run" is also given: • "none" = No fault occurred during this recording • "now" = The fault is there now • "prev" = The fault was there previously
Overrun	16	Overrun happened right before this value.

The protocol reserves the values 1 and 4 for 'Unknown' and 'Settling', but those are not supported by LAN-XI.

8.2.4.4 *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.[3] The number of signals in the message must be greater than zero
Reserved	Int16		Reserved. Set to 0

The following structure is then repeated "NumberOfSignals" times:

Name	Stream type	REST type	Description
SignalID	Int16	Number	Identifies the signal, i.e. where the data originates from
NumberOfValues[4]	Int16	Number	Number of values included in this message

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 Number	A single CAN message, see 0 for the contents of this data message type

8.2.4.4.1 *CAN message*

CAN messages are transmitted as *AuxSequenceData* described in previous section 8.2.4.4. The *values* of *AuxSequenceData* is an array where each element is an object containing CAN data as shown in the table below.

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

[4] An AuxSequenceData message may be sent with "NumberOfValues" set to 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 indicate that there are no more data at or before the given time (in BK Connect signal analysis parlance this is called a Synchronade).

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=11bit 1=29bit) 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 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 bytes	11-bit or 29-bit CAN message ID.
CanData	8 bytes	CAN data. Number of valid data is defined by “CanDataSize” field.

The status field contains important information about CAN-bus state and message errors. It can be compared to overload info for analog data. The first two bits contain information about CAN-bus state and are 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) contains info whether a RX/TX error has been 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 WebXI 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 bytes 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 bytes CAN data including padding

The green color in the example shows Web-XI header, the blue color shows *AuxSequenceData* header and brown color shows *AuxSequenceData* values that encapsulate CAN message.

8.2.4.5 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 32-bit word
Value	Depends on descriptor type	The value of the descriptor. This value must be a multiple of 32-bit word

8.2.4.6 *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 on each value in signal data to obtain a value in the specified unit.[5]	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 Strings 8.2.3.1	Empty string
VectorLength	6	Length of one value. 0 means scalar.	Int16	0
ChannelType	7	ChannelType enum	Int16	1

8.2.4.7 *ChannelType enum (Int16):*

The ChannelType (Int16) enum has the following possibilities:

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

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

8.2.4.8 *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	UTF8 string. Content is an Int16 length followed by the given number of bytes. See Strings 8.2.3.1

9 General concepts

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

9.1 REST

REST is an abbreviation of 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 to flip the switch.

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

9.2 HTTP

The REST protocol uses the standard HTTP protocol for communication. The LAN-XI device includes a web server that reacts to the REST commands and calls the relevant subsystems.

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
```

9.2.1 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.

9.2.1.1 Successful operation

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).

9.2.1.2 *Client errors*

In general, any 4xx 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.

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.

9.2.1.3 *Server errors*

5xx 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 Is 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.

9.3 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.

9.3.1 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 http://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
        }
}
```

9.3.2 XML

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

More information can be found in this tutorial: http://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 5.3.5. 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.

9.3.3 Other

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

9.4 State machine

The 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 application will not be possible at this point.

9.5 Time

The module operates with two types of time

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

9.6 License

The LAN-XI license-scheme is based on the so-called "Message-Authentication-Code" (MAC) scheme, where the license can be seen in clear text. This makes it easy for people involved to see which license belongs to which module. The clear-text part contains:

- Module Type (not subtype)
- Serial No
- License text (E.g. "Sonoscout")

Here is an example with three licenses for a 3057 Bridge Module with Serial No 105224:

```
3057-105224-Sonoscout-BZ-5950-L-N01 Sonoscout-
TF/q7iZ4Hd8PtXZphGno0xkKW0jMM1puozoyFKraTlfMUH3CAY0qGmqzB/jC1/O/dpvClpEyz/L9
n2pMk56SNQ==
3057-105224-InputStreaming-BZ-5951-L-N01 LAN-XI input channel streaming-
Fl7RyxjDNcflsp+Tt6U1nHXNn6rG+KwkbhTqAZ921RW1tJPm5a1dYBb3y6mAGX5IPbqWzJ1O8
MNsffVw2952jg==
3057-105224-Recorder-
rt6KyLHmloaMTzW8Wgy2f9e0pbUJJs0qqwXYAKZub5SScLKUmiL+/22a3Lk17/BeGWxYZ4K7Z
wBfKiM4dUNwDw==
```

10 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 – E.g. Pascal. The conversion is described as:

*calibratedValue = sampleValue * 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 * headroom / sensitivity*

headroom is a factor used by the DSP 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 0.00918V/Pa you get:

*scaleFactor = 10 V * 1.18850 / (0.00918 V / Pa) ~= 1295 Pa*

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