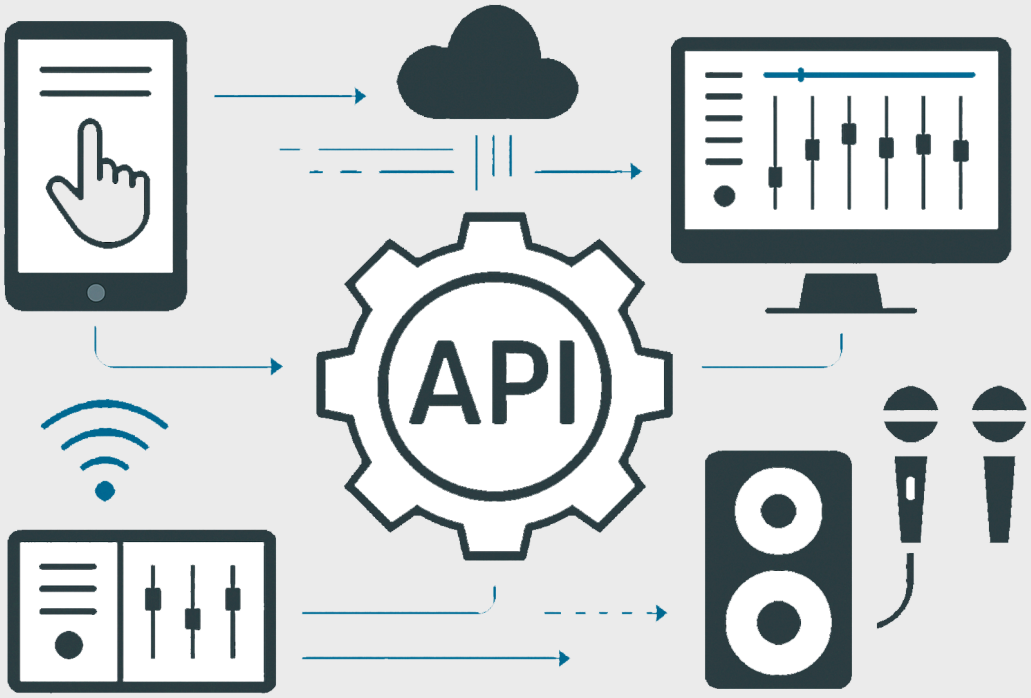


Sennheiser 3rd Party API

PDF export of the original HTML manual

Contents

1. Preface ... 3
2. Release Notes ... 4
3. Sennheiser Sound Control Protocol ... 5
    Sennheiser Sound Control (SSCv2) Specification - Version 2.3 ... 6
    Introduction ... 7
    SSC Overview ... 8
    Conventions ... 12
    SSCv2 Data Structure Specification ... 14
    General SSCv2 Address Schema ... 26
    SSCv2 Guidelines ... 44
    Sennheiser Sound Control Protocol v1 ... 51
4. OpenAPI ... 52
    Evolution Wireless Digital (EW-DX) ... 52
        EW-DX OpenAPI 1.7 ... 52
    TeamConnect Bar (TC Bar S/M) ... 53
        Known Issues ... 53
        TC Bar OpenAPI 1.16 ... 54
    TeamConnect Ceiling Medium (TCC M) ... 55
        TCC M OpenAPI 1.9 ... 55
        Subscription for real-time updates ... 56
    Spectera ... 58
        Introduction to Spectera API ... 58
        Usage of Spectera API ... 65
        Spectera OpenAPI specification versions ... 90
    MobileConnect ... 112
        MobileConnect OpenAPI ...

1. Preface

PDF Export of the Original HTML Manual

This PDF document is an automatic export of an interactive set of HTML manuals. Some content and interactive elements may not be included in the PDF because they cannot be displayed in this format. In addition, automatically generated page breaks may cause related content to be slightly shifted. We can therefore only guarantee the completeness of the information in the HTML manual and recommend using it. You can find it in the Documentation Portal at www.sennheiser.com/documentation.

2. Release Notes

Latest release information on the firmware and OpenAPI versions.

New Release

- 26-10-28 | New OpenAPI version 18.3 for Spectera is available (see Release Notes Version 18.3).

Previous Releases

- 26-07-09 | New OpenAPI version 18.1 for Spectera is available (see Release Notes Version 18.1).
- 26-07-06 | New OpenAPI version 1.16 for TC Bar S/M is available (see TC Bar OpenAPI 1.16).
- 26-03-25 | The first OpenAPI version 17.0 for Spectera is available (see Supported API versions).
- 26-02-25 | New OpenAPI version 1.9 for TeamConnect Ceiling Medium is available (see TCC M OpenAPI 1.9).
- 26-02-02 | New OpenAPI version 1.13 for TC Bar S/M is available (see TC Bar OpenAPI 1.16).
- 25-12-01 | New OpenAPI version 1.12 for TC Bar S/M is available (see TC Bar OpenAPI 1.16).
- 25-08-25 | The new OpenAPI version 1.8 for TCC M is available
- 25-07-08 | The first EW-DX OpenAPI is available.
- 25-06-02 | New OpenAPI version 1.11 for TC Bar S/M is available.
- 25-02-21 | New version of the SSC Protocol for Evolution Wireless Digital devices available.
- 24-08-15 | New OpenAPI version 1.7 for TCC M is available.
- 24-07-03 | The first OpenAPI version for TC Bar S/M is available.
- 23-07-01 | The TCC M firmware version 1.2.x is supporting the API 1.6.
- 22-05-05 | MobileConnect now provides plug-ins for Crestron and Extron 3rd party integrations, based on the MobileConnect API.

3. Sennheiser Sound Control Protocol (SSC)

All product specific HTTPs methods, parameters and responses at a glance.

> **i** Sennheiser offers two different protocols that can be used depending on the functional scope of the implemented device firmware and software supplied with the product:
>
> - **SSCv2:** New protocol with a high security standard for Sennheiser devices that are delivered with a password.
> - **SSCv1:** Old protocol with command and control functions without support for networked audio streaming.

SSCv2

The newest Sennheiser 3rd party API protocol allows to configure and monitor devices using REST API calls. The following Sennheiser devices are supported:

- Spectera
- TeamConnect Ceiling Medium (TCC M)
- TeamConnect Bar (TC Bar)
- Evolution Wireless Digital (firmware ≥ 4.0.0)
    - EW-DX EM 2 rack receiver (EW-DX EM 2)
    - EW-DX EM 2 rack receiver Dante (EW-DX EM 2 Dante)
    - EW-DX EM 4 rack receiver Dante (EW-DX EM 4 Dante)

SSC

The older protocol allows to command and to control devices. This protocol does not support the network audio streaming. The following Sennheiser devices are supported:

- SL Rack Receiver
- CHG 4N - network-enabled charger
- CHG 2N - 2 bay network charger
- Multi-channel receiver (SL MCR2 & MCR4)

Evolution Wireless Digital (firmware < 4.0.0)

- EW-DX EM 2 rack receiver (EW-DX EM 2)
- EW-DX EM 2 rack receiver Dante (EW-DX EM 2 Dante)
- EW-DX EM 4 rack receiver Dante (EW-DX EM 4 Dante)
- CHG70N - 2 bay network charger

- TeamConnect Ceiling 2 (TCC 2)

Sennheiser Sound Control (SSCv2) Specification - Version 2.3

Abstract

Specification for a secure, extendible, standards-based, versatile restful API via https transport, suitable for command, control, monitoring, metering, and scripting of networked audio devices; Utilizing JavaScript Object Notation (JSON) for data serialization.

History

- 2022, August 2: Version 0.1 (Draft)
- 2022, September 9: Version 0.8 (Draft)
- 2022, October 10: Version 2.0
- 2022, December 2: Version 2.1
- 2023, March 1: Version 2.2
- 2023, October 27: Version 2.3

Introduction

Modern professional audio devices are designed as building blocks for large, complex systems and are more and more integrated into the IT infrastructure of our customers. Controlling and monitoring the audio devices using IP-based standard protocols is mandatory for easy integration by third parties, which cannot be expected to learn a new protocol for every vendor. When exposing an audio device over the network security considerations to prevent malicious actors from misconfiguring, damaging or subverting the audio device must be taken into account. These considerations are mandatory to protect the device and the network of the customer.

With the dominating success of HTTP as the backbone of the Internet, HTTP and the principle of RESTful APIs are more and more dominating on-premise automation solutions. As HTTP addresses the issue of authentication and encryption of the network communication in a very adversarial environment, the secured variant, HTTPS, is a very mature and trusted solution for secure, standardized data exchange.

When considering the needs for an IP-based, standards-compliant solution it is imminent, that these needs are not compatible with the needs of inter-process communication, low-throughput links or configuration file formats. The following points are key for an IP-based communication protocol, which is exposed to be used by customers

- Standards compliant for transport and data formats to lower the bar of entry
- Standards compliant for employed security mechanisms
- Extensible/adaptable to allow for new product requirements
- Possible to be easily integrated in an existing network infrastructure

This document describes the specific adaption of HTTPS and the principle of RESTful APIs to Sennheiser use, "Sennheiser Sound Control v2", SSCv2. The main other ingredient is JavaScript Object Notation (JSON), which has become the default data encoding used in HTTPS-based RESTful APIs.

Note that the protocol is intended for command and control over IP-based networks only. Network audio streaming, inter-process communication, communication over low-throughput links, usage as file format are entirely out of its scope.

SSC Overview

The following standards are important for the understanding of SSCv2, as they form the basic building blocks the protocol is built upon.

HTTPS Overview

The Hypertext Transfer Protocol (HTTP) is the foundational protocol for data exchange in the World Wide Web. It is designed as an application-level protocol for distributed, collaborative, hypertext information systems. Finalized as version 1.0 in 1996 and updated to version 1.1 in 1997, HTTP has been the backbone of the World Wide Webs success since more than 20 years. It was updated to version 2 in 2015 and 3 in 2022. HTTP/3 changes the underlying transport protocol from TCP to UDP, while still retaining session semantics. From a user perspective there is no change in perceived session attributes, just an increase in throughput and a reduced latency. While these updates are enhancing the protocol, usage of 1.1 is still widespread.

To add security to the HTTP/1.1 protocol Transport Layer Security (TLS) is used, which adds security without changing the protocol itself, the resulting secure protocol is commonly referred as HTTPS.

HTTP/1.1 is standardized in the RFCs 7230 to 7240 (https://www.ietf.org/rfc/rfc7230.txt).

The SSC specification mandates the use of HTTP/1.1 and TLS 1.2.

Server Sent Events

HTTP is based on the Request-Response-Pattern, meaning every answer from the server must be triggered by a request by the client. The response happens immediately. Although there are some approaches with delayed answer (e.g. the server sending a reply indicating it is not finished until the answer is ready), this still follows the Request-Response-Model. To allow for unsolicited messages after an initial subscription Server Sent Events (SSE) are specified. They define a mechanism for clients to tell the server they want to be able to receive asynchronous messages and for the server to send these to them without a dedicated request for each response.

Server Sent Events are developed as a living standard by the Web Hypertext Application Technology Working Group (WHATWG). It can be found at: https://html.spec.whatwg.org/multipage/server-sent-events.html#server-sent-events

JavaScript Object Notation Overview

JavaScript Object Notation (JSON) is a lightweight data-interchange format. It is derived from the object literals of JavaScript, as defined in the ECMAScript Programming Language Standard, Third Edition.. It is based on a subset of the JavaScript Programming Language, Standard ECMA-262 3rd Edition - December 1999. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of all languages, including C, C++, C#, Java, JavaScript, Ruby, Python, and many others. These properties make JSON an ideal data-interchange language.

The central website for JSON information is http://json.org. JSON is formally specified in RFC 8259 (http://www.ietf.org/rfc/rfc8259.txt).

JSON's design goals are to be minimal, portable, textual, and a subset of JavaScript.

RESTful API Overview

Representational State Transfer (REST) is a software paradigm, which has become the main approach to create stateless, reliable web APIs. These web APIs are typically based on HTTP methods to access URLs and use JSON as data encoding.

A RESTful API defines resources (made accessible using an URI), encapsulating entities. A client is able to interact with the resources by accessing their URI. A big emphasis is put on the design of the API - the designer needs to carve resources and their entities in a way, which groups related entities in logical resources.

While RESTful is aiming to be completely stateless, the SSC protocol adds some state to its API (for example to allow subscriptions).

The initial definition of the REST principle is from Roy Fieldings dissertation, found at https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

OpenAPI

The OpenAPI specification is a specification for machine-readable descriptions of RESTful web services. The descriptions can be used not only to document, but also for the generation of code providing or consuming the interface or to assist in automatic test creation. Created in 2010 as the open-source Swagger specification, it was moved over to the OpenAPI initiative in 2015 and renamed to OpenAPI specification in 2016. The specification has established itself as the main standard for describing RESTful web services.

The standardization of OpenAPI is coordinated on https://www.openapis.org/.

OpenAPI allows for API specification in either JSON or YAML format and provides converters to easily convert between the formats. For the examples in this specification the YAML format was chosen, as this is better suited to human exchange. When providing the OpenAPI description of a device as part of the API, the use of JSON is mandated, to have the API stick to one data exchange format.

Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP14/RFC 2119, "Key words for use in RFCs to Indicate Requirement Levels" (https://datatracker.ietf.org/doc/html/rfc2119).

HTTP related terminology is used as in RFC 9110 (https://datatracker.ietf.org/doc/html/rfc9110). JSON related terminology can be found in RFC 8259 (https://datatracker.ietf.org/doc/html/rfc8259). REST is not a formal standard, information can be found in this dissertation (https://roy.gbiv.com/pubs/dissertation/top.htm).

SSCv2 Terminology

Control resource

- Resource, which, in the context of dynamic resources, can be subscribed to to get an implicit subscription to all resources controlled by this resource; see Dynamic Resources.

SSCv2 Message

- JSON object that is either transferred as HTTP request/response or as server sent event (SSE)

SSCv2 Server

- device or application that receives SSCv2 messages via HTTP request, and replies to them

SSCv2 Client

- device or application that sends SSCv2 messages via HTTP request

SSCv2 Command

- HTTP request sent by an SSCv2 client (includes URI, Header)

SSCv2 Address

- Path component of the URI of an HTTP endpoint implemented by an SSCv2 Server; will be used synonymously to resource

SSCv2 Address Tree

- hierarchical tree comprising all the SSCv2 Addresses of an SSCv2 Server

SSCv2 Command Arguments

- JSON object sent as HTTP body and/or query argument of an SSCv2 command

SSCv2 Command Reply

- HTTP response sent by SSCv2 Server as reply to an SSCv2 command

SSCv2 Session

- association of a specific SSCv2 Client to an SSCv2 Server

SSCv2 Data Structure Specification

SSCv2 constraints of underlying protocols

Case sensitivity

Case sensitivity is a contested topic. SSCv2 is following the standards it uses in this regard:

- The path component of an URI MUST be implemented case sensitive, as per RFC 3986, 6.2.2.1, "Case normalization". An SSCv2 schema still SHOULD avoid specifying paths which are ambiguous on case.
- As JSON specifies its strings to follow C conventions and nearly all JSON implementations are case sensitive, the JSON payload MUST be interpreted by SSCv2 Clients and SSCv2 Servers as case sensitive. An SSCv2 schema still SHOULD avoid specifying entities which are ambiguous on case.

HTTP(S)

The following optional parts of HTTP(S) have been made mandatory or are recommended for SSCv2:

- An SSCv2 Server MUST implement HTTPS (=encrypted connections)
- An SSCv2 Server MUST NOT allow HTTP (=unencrypted connections)
- An SSCv2 Server MUST implement HTTP/1.1
- An SSCv2 Server SHOULD implement persistent connections to limit the amount of new connections needed
- An SSCv2 Client SHOULD use persistent connections

Note for development: An SSCv2 Server MAY implement a development build allowing for HTTP (e.g. to help with debugging), but MUST ensure that HTTP is not possible in release versions.

Default HTTP status codes

For some generic actions there have been HTTP status codes made mandatory:

- **400 (Bad Request)**: When a client is authorized to access a resource, but has a format error in its request, the SSCv2 Server MUST reply with 400 - "Bad request". Examples for error 400 are messages with broken JSON, but also messages, which violate limits for entities or use enum-values, which are not specified.
- **401 (Unauthorized)**: When a client sends a request without being authenticated (see "Specification for Authenticated Sennheiser Sound Control (SSC) Clients"), the SSCv2 Server MUST reply with 401 - "Unauthorized".
- **403 (Forbidden)**: When a client is authenticated, but not authorized to access the resource, the SSCv2 Server MUST reply with 403 - "Forbidden".
- **404 (Not Found)**: When a client is authorized and tries to access a resource which is not existing, the SSCv2 Server MUST reply with 404 - "Not Found".
- **405 (Not Allowed)**: When a client is authorized to access a resource, but the HTTP request method is not allowed by the SSCv2 Server, the SSCv2 Server MUST reply with 405 - "Method not allowed".
- **409 (Conflict)**: When a client is authorized to access a resource, the format checks out, but the device can not honor the request because of internal device state, the SSCv2 Server MUST reply with 409 - "Conflict". An example for this would be trying to configure an IP address, while the device is using DHCP, without also switching to use fixed addresses.
- **422 (Unprocessable Entity)**: When a client is authorized to access a resource, the format checks out, but there are semantic/logical errors, the SSCv2 Server MUST reply with 422 - "Unprocessable Entity". An example for a logical error would be using a sessionUUID of a subscription which does not exist.

This list is non-exhaustive and an SSC Server implementation is free to choose additional HTTP status codes if fitting.

An SSCv2 Server MUST first check for authentication, then for authorization. After that it is free to check in any order deemed fitting.

Error handling

An SSCv2 Server SHOULD add additional error arguments, if this helps to understand the error. The format of the arguments MUST be specified and explained in the OpenAPI schema.

JSON

The following optional parts of JSON have been made mandatory or are recommended for SSCv2:

- An SSCv2 Server MUST return all text-based entities as a JSON object. While JSON allows for singular values to be returned in a plain way, this is a newer addition to the JSON standard and might not be supported by all JSON libraries.

JSON Message Transaction Syntax

The SSCv2 Message exchange is described here as transaction using the following syntax:

- Prefix „TX:“ indicates an SSCv2 Message an SSCv2 Client is sending to an SSCv2 Server. There are up to four parameters to a TX:
    - HTTP verb (GET, PUT, POST are used in these examples)
    - HTTP resource identifier (Path)
    - HTTP query parameters
    - optional HTTP body (if present, encoded as JSON object)
- Prefix „RX:“ indicates an SSCv2 Message that the SSCv2 Server will send back to the Client. It consists of two parameters:
    - HTTP status code
    - optional HTTP body (if present, encoded as JSON object)

Getter commands, which request a property from the SSCv2 Server, are realized by using the HTTP GET verb. A GET SHOULD always be replied with the complete object in JSON format using a Content-Type of application/json. Exceptions, like returning binary data, MAY be chosen, but MUST be justified and document why it is impossible to use JSON. The SSCv2 Server MUST set the correct Content-Type accordingly.

Read

```
TX:
  Verb: GET
  Path: api/out1/xlr2
RX:
  Status Code: 200
  Body: {"gain": -10, "mute": false}
```

A transaction to set the gain of "xlr2" of "out1" to -10 looks like this:

```
TX:
  Verb: PUT
  Path: api/out1/xlr2
  Body: {"gain": -10}
RX:
  Status Code: 200
```

Note that the command results in an empty reply message using the status code 200 ("ok"). The SSCv2 Server MAY specify a reply containing a body, if the result of changing the entity is not the verbatim input data (e.g. a new resource is created based on the input data). Although the resource identifier `out1/xlr2` may consist of more parameters, only the `gain` was modified. It is also possible to modify multiple parameters of a property:

```
TX:
  Verb: PUT
  Path: api/out1/xlr2
  Body: {"gain": -10, "mute": false}
RX:
  Status Code: 200
```

The input MUST NOT be converted automatically if it is found in violation to the expected input. Any invalid access, e.g., writing to a read-only property, or if the argument is out of

range, MUST result in an error. The SSCv2 Server MAY add additional information explaining the error to the error message's body.

```
TX:
  Verb: PUT
  Path: api/out1/xlr2
  Body: {"gain": -1000}
RX:
  Status Code: 400
  Body: TBD
```

An SSCv2 Server MUST implement setting multiple parameters as an transactional operation; This means a partial failure of a command MUST lead to the SSCv2 Server not applying any of the contained arguments.

```
TX:
  Verb: GET
  Path: api/out1/xlr2
RX:
  Status Code: 200
  Body: {"gain": -10, "mute": false}
TX:
  Verb: PUT
  Path: api/out1/xlr2
  Body: {"gain": -5, "mute": filenotfound}
RX:
  Status Code: 400
  Body: TBD
TX:
  Verb: GET
  Path: api/out1/xlr2
RX:
  Status Code: 200
  Body: {"gain": -10, "mute": false}
```

Note that the `gain` parameter was not changed to `-5` since the requested `mute` parameter was invalid.

```
TX:
  Verb: POST
  Path: api/device/restart
RX:
  Status Code: 200


TX:
  Verb: POST
  Path: api/device/restore
  Body: {"type": "Factory Defaults"}
RX:
  Status Code: 200
  Body: {"type": "Factory Defaults"}
```

SSCv2 JSON Message Syntax

SSCv2 Messages

A Message is the protocol unit of transmission. There are two ways of message exchange:

- Synchronous: The client sends a request and the server immediately sends a response
- Asynchronous: The client subscribes for parameter changes and the server notifies the client on parameter changes via SSE.

An SSCv2 Message MUST be sent as a single closed JSON form describing a JSON object. This means that every SSCv2 Message is enclosed in a pair of curly brackets `{ }`. There MUST NOT be more than one SSCv2 message per HTTP request.

SSCv2 Addresses

Every SSCv2 Server implements a set of SSCv2 Addresses. SSCv2 Addresses are the potential destinations of SSCv2 Messages received by the SSCv2 Server, and correspond to each of the points of control that the device makes available. The SSCv2 Server MUST respond to each received SSCv2 Command by sending an SSCv2 Command Reply to the originating SSCv2 Client.

Each SSCv2 Address is addressable by a unified resource identifier (URI). An SSCv2 Server's SSCv2 Address are arranged in a tree structure called an SSCv2 Address Tree. The leaves of this tree are the SSCv2 Addresses. An SSCv2 Server's SSCv2 Address Tree MAY be dynamic; that is, its nodes and leaves MAY change over time.

The SSCv2 Address follows the URI specification, but MUST NOT use percent encoding. Hence only US-ASCII MUST be used to form an SSCv2 Address. While there is no limit in HTTP to the length of an URL and it recommends to support at least 8000 chars, an SSCv2 Address SHOULD not exceed 2048 characters for practicality purposes.

Temporal Semantics

By default, the SSCv2 Server SHOULD execute the SSCv2 Command immediately, i.e., as soon as possible after receipt of the message.

An SSCv2 Server MAY expose a representation of its current absolute device time. The optional SSCv2 Command `/device/time` MAY be used to query and optionally set the device time. SSCv2 does not provide a mechanism for clock synchronization; if an SSCv2 Server utilizes a mechanism like NTP or PTP to sync to the absolute time it MUST implement `/device/time` as read-only parameter. See also the description of this resource in General SSCv2 Address Schema.

Commands with long execution times

If the SSCv2 Server expects the execution of a command to exceed 5 seconds it MUST immediately send an HTTP response with status code `102 - processing`. When the command execution is finished the SSCv2 Server finalizes the request by sending the response containing the final result and appropriate status code (e.g. `200 - ok` or corresponding error).

SSCv2 Methods addressing array values

A resource MAY have an array of elementary data as entity. The array MAY be empty. The SSCv2 Command acting upon the resource MUST NOT switch between returning arrays and elementary data.

All the elements of the array MUST have the same elementary data type.

In previous version of this standard the access of array (special) ranges was required, this SHOULD NOT be used anymore.

Accessing complete arrays

Reading an array is done in the same way as reading an elementary value.

Writing an array is done in the same way as writing an elementary value with the array as payload.

```
TX:
  Verb: GET
  Path: api/presets/bank1
RX:
  Status Code: 200
  Body: { "carriers": [470000,470400,470800,471200,471600] }
TX:
  Verb: PUT
  Path: api/presets/bank1
  Body: { "carriers": [470000,470400,470800,471250,471600] }
RX:
  Status Code: 200
```

SSCv2 Sessions

An SSCv2 Session is defined by the association of a specific SSCv2 Client with an SSCv2 Server. The SSCv2 Server MUST keep state information specific to each SSCv2 Client (e.g., state relating to SSCv2 subscriptions, or authorization). The SSCv2 Server MUST couple the state to the SSCv2 Session. When the SSCv2 Session terminates, session state MUST be cleared (e.g., all SSCv2 subscriptions of the client are cancelled).

An SSCv2 Session begins with establishing the HTTPS connection from a specific SSCv2 Client to an SSCv2 Server.

The SSCv2 Session MUST last for the duration of the connection. The SSCv2 Session MUST be terminated when the connection is closed by either side.

The SSCv2 Server SHOULD keep state information that is explicitly specific to the SSCv2 Session in SSCv2 Addresses that are below `/api/ssc/state`, e.g., information about SSCv2 subscriptions (see also section SSCv2 subscriptions - /api/ssc/state/subscriptions). Additionally, SSCv2 Server state specific for an SSCv2 Session MAY affect the results of SSCv2 Commands (e.g. it might reject specific SSCv2 Commands if an SSCv2 Client lacks sufficient access permission).

The SSCv2 Server MUST NOT coalesce SSCv2 Sessions together based on authorization credentials. Multiple SSCv2 sessions per authorization credential MUST be the default.

General SSCv2 Address Schema

The SSCv2 Address Tree MUST be located at /api. This is mandatory, so an SSCv2 implementation is able to be incorporated into a web server setup easily if needed.

Some parts of the SSCv2 Address Tree are mandated by this specification for purposes of meta-protocol information, generic device-independent features, and device capability description. Any SSCv2 Server MUST implement these addresses. The reserved parts of the address tree are rooted in the SSCv2 Addresses: /api/ssc /api/device

The addresses and commands rooted in these reserved addresses are described in the following sections. This specification should be revised if additional addresses in the address tree are made mandatory.

The schema descriptions given in the following are written in the YAML notation of the OpenAPI Specification (OAS). This format allows a very detailed and structured description of the HTTP verbs, the addresses, the payload and the responses. Nevertheless, in this document only the most important points of the address schema are covered for the sake of readability. Therefore, for some addresses the responses might be shortened and not covering all edge cases or error messages.

SSCv2 Protocol version - /api/ssc/version

Read-only value. Reports the SSCv2 protocol version and the schema version implemented in the server.

```yaml
/api/ssc/version:
  get:
    summary: Get the versions relevant for SSCv2 implemented
    description: An SSCv2 Command to retrieve the SSCv2 protocol and schema versions
 implemented in the SSCv2 Server
    responses:
      '200':
        description: Successful request
        content:
          application/json:
            schema:
              type: object
              properties:
                protocol:
                  type: string
                  example: 2.0
                schema:
                  type: string
                  example: 1.5
      '500':
        description: SSCv2 Server encountered internal error
```

Session-specific SSCv2 Address Space - /api/ssc/state

SSCv2 Addresses under the `/api/ssc/state` address are specific to the SSCv2 Session between SSCv2 Client and SSCv2 Server. This means that it is possible that different SSCv2 Clients invoke the same SSCv2 Command with different arguments, and the immediate reply as well as the resulting state of the SSCv2 Server will differ for each SSCv2 Client.

This behavior differs from the normal behavior of an SSCv2 Server, where the server state is shared between all SSCv2 Clients.

SSCv2 subscriptions - /api/ssc/state/subscriptions

The SSCv2 Server MUST implement subscriptions using Server Sent Events (SSE), formatting messages according to the EventSource specification. From the EventSource specification only the parameters `data` and `event` are allowed to be used. For event only the following events MUST be used:

- open, on the start of a subscription
- message, when sending resource updates. As this is the default, this MAY be omitted in the event stream
- close, when the SSCv2 client actively closed the subscription using DELETE on `/api/ssc/state/subscriptions/{sessionUUID}` or the SSCv2 Server closes the subscription as a side effect of an SSCv2 Command.

An SSCv2 Server MUST send a close-event, when the subscription is closed by the Server and the Client is still present, e.g. when initiating a reboot or changing the ip address used by the subscription.

A full EventSource compatible update looks like the following:

```
event: <eventtype>\ndata: <updatejson>\n\n
```

`\n` denotes the ASCII-character for a linefeed.

The subscription is initialized by the client by issuing a GET request. The SSCv2 Server either accepts or refuses the subscription request based on the credentials used. When accepting the SSCv2 Server MUST reply with a message setting the `Content-Type` to `text/event-stream` and the header field `Content-Location` containing a the path `/api/ssc/state/subscriptions/{sessionUUID}` set to an internally generated ID on subscription and associated with the HTTP session. The SSCv2 Server MUST sent a first message to the client, containing the initial open-event. The SSCv2 subscription starts empty without any subscribed resource and the SSCv2 Client is able to subscribe to resources using `/api/ssc/state/subscriptions/{sessionUUID}` and `/api/ssc/state/subscriptions/{sessionUUID}/add` respectively.

The SSCv2 Server MAY refuse subscription requests, subject to device-specific policy or implementation specific limitations. The SSCv2 Server MUST reply on the subscription request immediately either by acknowledging the request, or by sending an error reply.

Each subscription notification MUST be equal to a GET request to the resource without any parameters.

An SSCv2 Server MUST treat subscriptions to resources not specifying GET requests as an error. If the user is authenticated, but not authorized to access a resource, the SSCv2 Server MUST treat this as an error.

An SSCv2 Server MUST NOT allow usage of array ranges for `/api/ssc/state/subscriptions`.

An SSCv2 Server MUST add the created resource URL (`/api/ssc/state/subscriptions/{sessionUUID}`) to the Content-Location-Header of the 200 - "OK" reply.

An SSCv2 Server SHOULD send out notifications as soon as possible.

An SSCv2 Server MUST generate the sessionUUID in a way which prevents guessing already existing sessionUUIDs - for example by using hashes of at least 8 Bytes of random or implementing RFC4122 (https://www.rfc-editor.org/rfc/rfc4122.html).

An SSCv2 Server MUST ensure that all accesses to a subscription (getting status, add, remove, replace) are using same access credentials as the initial creation of the subscription. In case of mismatching credentials the SSCv2 Server MUST reply with 403 - Forbidden, without processing the request.

An SSCv2 Server MUST send immediately the initial open-event, formatting the attached data according to the following json-template. `{sessionUUID}` MUST be replaced with the UUID of the newly initialized session.

```
{
  "path": "/api/ssc/state/subscriptions/{sessionUUID}",
  "sessionUUID": "{sessionUUID}"
}
```

An SSCv2 Client MUST NOT send further regular HTTP requests to the SSCv2 Server using the connection used to request the subscription. After the initial subscription request the connection can only be used to receive events from the SSCv2 Server. The response header reports the sessionUUID for later relation to modify of the subscription and the body is the stream of the values of the subscribed properties at the time of subscription and all the following updates in the time during the subscription as shown in OpenAPI example below. For an example of the answer of the stream, see Section Subscription notification syntax below.

OpenAPI snippet

```
/api/ssc/state/subscriptions:
  get:
    summary: Start a subscription
    description: An SSCv2 Command to start a subscription
    responses:
      '200':
        description: Successful request
        headers:
          Content-Location:
            description: Location of the resource to be used to change the subscribed
 resources
            schema:
              type: string
        content:
          text/event-stream:
            schema:
              type: string
              description: The initial subscription stream.
      '401':
        description: Unauthorized since no user credentials are passed
      '403':
        description: User is not authorized to subscribe to resources
      '500':
        description: SSCv2 Server encountered internal error
```

Getting subscription status

If an SSCv2 Client wants to request the subscription status attached to a `sessionUUID`, it MUST use a GET request to `/api/ssc/state/subscriptions` using this `sessionUUID`. An SSCv2 Server MUST check that the permission of the SSCv2 Client requesting is adequate for requesting the subscription list associated with the `sessionUUID`. For example with an control user and an API user, the control user SHOULD be allowed to request the subscription list for any user, while the API user is only allowed to get lists created with the API user. The different users are specified in the "Specification for Authenticated Sennheiser Sound Control (SSC) Clients".

An SSCv2 Server MUST ensure that this subscription has been created by the same user as the user sending the request.

An SSCv2 Server MUST treat a non-existing `sessionUUID` as an error.

OpenAPI snippet

```
/api/ssc/state/subscriptions/{sessionUUID}:
  get:
    summary: Get the subscription list
    description: An SSCv2 Command to retrieve the list of subscriptions associated with
 the sessionUUID
    parameters:
        - in: path
          name: sessionUUID
          schema:
            type: string
          required: true
    responses:
      '200':
        description: Successful request
        content:
          application/json:
            schema:
              type: array
              items:
                type: string
                example: /api/device/site
      '401':
        description: Unauthorized since no user credentials are passed
      '403':
        description: User is not allowed to get subscription list for this sessionUUID
      '422':
        description: sessionUUID did not exist
      '500':
        description: SSCv2 Server encountered internal error
```

Changing subscribed resources

It is possible to either change the full set of resources of a subscription, or to add/remove resources from the currently subscribed set.

Changing the full set of subscribed resources

To change the full set of subscribed resources, the SSCv2 Client MUST send a complete list of the resources it wants to be subscribed to after changing the subscription to `/api/ssc/state/subscriptions/{sessionUUID}`. Akin to getting the subscription status an existing subscription is updated by using the `sessionUUID` query parameter. The SSCv2 Server MUST treat updating the list as a transaction and MUST check that the permission of the SSCv2 Client requesting is adequate for changing the subscription list associated with the `sessionUUID`. The SSCv2 Server MUST remember that the requesting client is to be notified about value changes of the subscribed addresses. After accepting and setting up the subscription the SSCv2 Server MUST send the current state of each subscribed resource which is new to the set of subscribed resources.

An SSCv2 Server MUST treat a non-existing `sessionUUID` as an error.

An SSCv2 Server MUST NOT treat an empty subscription list as cancelling and MUST NOT terminate the subscription.

If the subscription is extended to include more addresses, the initial values of the newly added and removed resources MUST be reported as for a new subscription.

OpenAPI snippet

```
/api/ssc/state/subscriptions/{sessionUUID}:
  put:
    summary: Set/change the subscription list
    description: An SSCv2 Command to set/change the list of subscriptions associated
 with the sessionUUID
    parameters:
        - in: path
          name: sessionUUID
          schema:
            type: string
          required: true
    requestBody:
        content:
          application/json:
            schema:
              type: array
              items:
                type: string
                example: /api/device/site
    responses:
      '200':
        description: Successful request
      '400':
        description: The request for subscription was invalid
        content:
            application/json:
              schema:
                type: object
```

```
              properties:
                path:
                  type: string
                  example: /api/ssc/version
                error:
                  type: integer
                  example: 403
    '401':
      description: Unauthorized since no user credentials are passed
    '403':
      description: User is not allowed to change subscription list for this
sessionUUID
    '422':
      description: sessionUUID did not exist
    '500':
      description: SSCv2 Server encountered internal error
```

Adding resources to subscriptions

To add one or more resources to an existing list of subscribed resources, an SSCv2 client MAY use the resource `/api/ssc/state/subscriptions/{sessionUUID}/add`. The SSCv2 Server MUST check, whether the SSCv2 Client is authorized to access these resources, if this subscription has been created by the same user, and whether these resources exist. If these checks succeed, the SSCv2 Server MUST add the resources to the list of subscribed resources of the existing subscription.

If one resource in the set of resources the SSCv2 Client wants to add is not allowed, the SSCv2 Server MUST refuse the complete request and the current set of subscribed resources for the subscription MUST be unchanged.

An SSCv2 Server MUST treat a non-existing `sessionUUID` as an error.

An SSCv2 Server MUST treat an empty resource list as no action and reply with 200 - OK.

An SSCv2 Server MUST ignore a resource already in the list of subscribed resources. If all resources in the resource list are already in the list of subscribed resources, the SSCv2 Server MUST still reply with 200 - OK.

An SSCv2 Server MUST report the initial values of the newly added resources using the existing subscription, at the moment the subscription addition request is granted.

OpenAPI snippet

```
/api/ssc/state/subscriptions/{sessionUUID}/add:
  put:
    summary: Add resource(s) the subscription list
    description: An SSCv2 Command to add a set of resources to the list of subscriptions
 associated with the sessionUUID
    parameters:
        - in: path
          name: sessionUUID
          schema:
            type: string
          required: true
    requestBody:
        content:
          application/json:
            schema:
              type: array
              items:
                type: string
                example: /api/device/site
    responses:
      '200':
        description: Successful request
      '400':
        description: The request to add to the subscription was invalid
        content:
            application/json:
              schema:
                type: object
                properties:
                  path:
                    type: string
                    example: /api/ssc/version
                  error:
```

```
                  type: integer
                  example: 403
      '401':
        description: Unauthorized since no user credentials are passed
      '403':
        description: User is not allowed to add to the subscription list for this
sessionUUID
      '422':
        description: sessionUUID did not exist
      '500':
        description: SSCv2 Server encountered internal error
```

Removing resources from subscriptions

To remove one or more resources from an existing list of subscribed resources, an SSCv2 client MAY use the resource `/api/ssc/state/subscriptions/{sessionUUID}/remove`. The SSCv2 Server MUST check, whether these resources are currently subscribed and if this subscription has been created by the same user. If these checks succeed, the SSCv2 Server MUST remove the resources from the list of subscribed resources of the existing subscription.

If one resource in the set of resources the SSCv2 Client wants to remove, is not currently subscribed, the SSCv2 Server MUST refuse the complete request and the current set of subscribed resources for the subscription MUST be unchanged.

An SSCv2 Server MUST treat a non-existing `sessionUUID` as an error.

An SSCv2 Server MUST treat an empty resource list as no action and reply with 200 - OK.

An SSCv2 Server MUST NOT terminate the subscription, if the list of subscribed resources is empty after the removal of resources.

The only error-value allowed in responses is 404, meaning "resource not in list of subscribed resources".

OpenAPI snippet

```
/api/ssc/state/subscriptions/{sessionUUID}/remove:
  put:
    summary: Remove resource(s) from the subscription list
    description: An SSCv2 Command to remove a set of resources from the list of
 subscriptions associated with the sessionUUID
    parameters:
        - in: path
          name: sessionUUID
          schema:
            type: string
          required: true
    requestBody:
        content:
          application/json:
            schema:
              type: array
              items:
                type: string
                example: /api/device/site
    responses:
      '200':
        description: Successful request
      '400':
        description: The request to remove to the subscription was invalid
        content:
            application/json:
              schema:
                type: object
                properties:
                  path:
                    type: string
                    example: /api/ssc/version
                  error:
                    type: integer
```

```
                  example: 404
      '401':
        description: Unauthorized since no user credentials are passed
      '403':
        description: User is not allowed to remove from the subscription list for this
sessionUUID
      '422':
        description: sessionUUID did not exist
      '500':
        description: SSCv2 Server encountered internal error
```

Subscription cancelling

An SSCv2 Client can end a subscription either implicitly or explicitly.

- To end a subscription implicitly, the SSCv2 Client just closes the connection receiving the subscription data.
- To end a subscription explicitly, an SSCv2 Client uses the DELETE request method to the resource `/api/ssc/state/subscriptions/{sessionUUID}`. When using the explicit method, the SSCv2 Server MUST send a close-event to the subscription, before it closes the connection. The data sent in the close-event is the same as in the open-event.

The SSCv2 Server MUST terminate a subscription in these cases:

- the subscribed client cancels the subscription explicitly
- the SSCv2 Client closes the connection
- the transport layer of the SSCv2 connection signals a fatal communication error
- the password of the user used to create the subscription was changed

When a subscription is closed, an SSCv2 Server MUST remove the associated sessionUUID.

When a subscription is closed explicitly, the SSCv2 Server MUST ensure that this subscription has been created by the same user as the user sending the request.

An SSCv2 Server MUST ensure to only close the subscriptions created using the credentials whose password has been changed. An SSCv2 Server MUST close subscriptions regardless whether the password was actually changed or the already present password was set again.

OpenAPI snippet

```
/api/ssc/state/subscriptions/{sessionUUID}:
  delete:
    summary: End an existing subscription
    description: An SSCv2 Command to end the subscription associated with the
 sessionUUID
    parameters:
        - in: path
          name: sessionUUID
          schema:
            type: string
          required: true
    responses:
      '200':
        description: Successful request
      '401':
        description: Unauthorized since no user credentials are passed
      '403':
        description: User is not allowed to end subscription for this sessionUUID
      '422':
        description: sessionUUID did not exist
      '500':
        description: SSCv2 Server encountered internal error
```

Subscribing to multiple addresses

The SSCv2 Client MAY request to subscribe to multiple resources in a single request by design.

The SSCv2 Server MUST treat all those subscription requests as one transaction. If the SSCv2 Server is not able to satisfy subscription to one of the resources it MUST send an error and refuse all resources. If there are already subscribed resources present, the previous subscribed state MUST remain unchanged.

Error handling

When the SSCv2 Server refuses a request for a subscription or a set of subscriptions, it MUST return an object containing the resource which was not subscribeable and the reason for the error. The SSCv2 Server MUST stop processing a request for a set of subscriptions after encountering the first invalid subscription in the request. The following error codes MUST be used:

- 403 if subscribing to the resource is not allowed for the user
- 404 if the resource does not exist

Subscription notification syntax

A subscription notification is JSON object as Server Sent Event (SSE) data, using the resource triggering the notification as a key and the entities of the resource wrapped in a JSON object as value. An SSCv2 Server MAY combine notifications for multiple resources in one JSON object, if the update occurs at the same time. Since a newly created subscription of multiple addresses MUST report the values of the resources this MAY be combined into a bigger JSON object. If this combination of the different resources is not supported, all notifications MUST be send in the stream as individual JSON objects.

The following example shows the stream of a notification after a initial subscription and a later notification of the change of the name:

```
{
    "/api/device/site":
  {
        "name": "MyDevice",
        "location": "Chemistry Building 5, Room 15",
        "site": "Left side near the big pillar"
    }
}
{
  "/api/device/site":
  {
    "name": "MyRenamedDevice",
    "location": "Chemistry Building 5, Room 15",
    "site": "Left side near the big pillar"
    }
}
```

The address `/api/device/site` was subscribed, the initial values were reported. After some time the `name` was changed and the resulting update was sent as a second JSON object.

Generic Device Information and Settings: Address Tree - /api/device

The device settings are parameters that are common to any compliant device on the network that are relevant to the device as a whole.

/api/device/identity

A read-only resource containing standard information about the device in question. The following entities are mandatory:

- product: Product identification, should be identical to the designation on the label on the product itself.
- serial: Unique product number, MUST be unique for this product.
- vendor: Vendor string, for Sennheiser-products it is „Sennheiser“.

OpenApi snippet:

```
/api/device/identity:
  get:
    summary: Get the device identity
    description: An SSCv2 Command to retrieve the device identity
    responses:
      '200':
        description: Successful request
        content:
          application/json:
            schema:
              type: object
              properties:
                product:
                  type: string
                  example: TeamConnect Ceiling 2
                hardwareRevision:
                  type: string
                  example: DVT
                serial:
                  type: string
                  example: AB12DEF345
                vendor:
                  type: string
                  example: Sennheiser electronic GmbH & Co. KG
                  enum:
                    - Sennheiser electronic GmbH & Co. KG
      '500':
        description: SSCv2 Server encountered internal error
```

/api/device/site

User-changeable values helping the user to individualize the device and the most convenient way for the customer to identify the device. The resource MUST contain and allow for configuration the following entities:

- deviceName: User-settable persistent device name. This name should be the preferred, and most convenient way for the customer to identify devices. If the device has a display, this name SHOULD be displayed there, preferably in the network context; if it has a menu, this name SHOULD be configurable or be derived from `/api/device/identity` or/and and the `location` entity of this resource. This name is to be understood as device identification in a network environment.
- location: User-settable persistent installation location, like e.g. `Chemistry Building 5, Room 15`. Free-form string entity.
- position: User-settable persistent position in the installation location in case there are multiple devices at the installation location. Free-form string entity.

The maximum length of string entities CAN be decided by a device and MUST be communicated using the OpenAPI schema for the device.

OpenApi snippet:

```
/api/device/site:
  get:
    summary: Get the device site information
    description: An SSCv2 Command to retrieve the device site information
    responses:
      '200':
        description: Successful request
        content:
          application/json:
            schema:
              type: object
              properties:
                name:
                  type: string
                  example: MyDevice
                location:
                  type: string
                  example: Chemistry Building 5, Room 15
                position:
                  type: string
                  example: Left side near the big pillar
      '401':
        description: Unauthorized since no user credentials are passed
      '403':
        description: User is not allowed to access this resource
      '500':
        description: SSCv2 Server encountered internal error
  put:
    summary: Write the device site information
    description: An SSCv2 Command to set the device site information
    requestBody:
      content:
        application/json:
          schema:
            type: object
            properties:
```

```
            deviceName:
              type: string
              maximum: 60
              example: MyDevice
            location:
              type: string
              maximum: 240
              example: Chemistry Building 5, Room 15
            position:
              type: string
              maximum: 240
              example: Left side near the big pillar
    responses:
      '200':
        description: Successful operation
      '400':
        description: Bad Request
      '401':
        description: Unauthorized since no user credentials are passed
      '403':
        description: User is not allowed to access this resource
      '500':
        description: Internal Server Error
```

SSCv2 Guidelines

Dynamic Resources

An SSCv2 Server MAY allow dynamic resources (for example allowing the user of SSCv2 to instantiate DSP modules on demand). The planning of the resources SHOULD follow a "Create, Read, Update, Delete"-approach (CRUD).

A good approach is to define a control resource, which allows for getting a full list of all instantiated dynamic resources associated with this resource (GET) and for creating new resources (POST). An SSCv2 Server MUST always return the newly created dynamic resource as Location-HTTP-header using a status code of 201 - "Created". An SSCv2 Client can then update (PUT) the newly instantiated dynamic resource using the location it received on creation or deinstantiate it using a DELETE request.

A resource controlled by a control resource is called a subresource.

The following rules are defined for control resources:

- The subresource MUST be one level below the control resource
- The subresource MUST be variable and specified by a singular template expression (e.g. variable , full path /api/mts/)
- There MUST NOT be more than one level of resources below a subresource
- Control resources MUST NOT be nested
- The control resource MUST be marked with the resource type ControlResource (see x-sennheiser-sscv2-resourcetype, OpenApi vendor extensions)
- The controlled subresource MUST be specified in the control resource (see x-sennheiser-sscv2-subresource, OpenApi vendor extensions)
- The elements of the control resource response content SHOULD contain a property which is identical (in both name and type/format) to the resource location path parameter of the subresources. This helps mapping the singular GET request on the control resource and the GET requests/subscription updates of the subresources.

Dynamic Resource Subscriptions

For Dynamic Resources subscriptions are handled in a special manner. A subscription to the the control resource will be recorded in the list of subscribed resources, but implicitly it is a subscription to all controlled subresources, currently existing and created later. A client will receive updates for all subresources and resources below them instead of changes of

the control resource. An SSCv2 Server implementing subscriptions for Dynamic Resources adheres to the following rules:

- A subscription to a control resource MUST only record the subscription to the control resource in the list of subscribed resources
- On subscription to the control resource the SSCv2 Server MUST send an update for all existing subresources controlled by this control resource, as well as for all resources below the subresources
- If a client subscribes to a control resource and explicitly to a specific subresource or a resource below a subresource, the SSCv2 Server MUST treat this as two different subscriptions and send two updates for the same resource on update

Dynamic Resource Creation

On creation of a dynamic resource the SSCv2 Server MUST notify the clients subscribed to the control resource. This is done by sending a notification to all clients subscribed to the control resource, using the subresource as key and the empty JSON object "{}" as value. This MUST then be immediately followed by an update with the current values of the subresource.

The following example shows the SSE data notifications for the creation of the resource /api/device/dsp/modules/agc/10

```
{
    "/api/device/dsp/modules/agc/10":  {}
}
{
    "/api/device/dsp/modules/agc/10":  { "value1": "a", "value2": "b", "value3": "c" }
}
```

If the subresource has resources below it, the creation notification MUST only be sent for the subresource. When the subresource is created, all resources below it are implicitly created.

Dynamic Resource Deletion

The SSCv2 Server MUST notify the clients subscribed to the control resource when a specific subresource is deleted. This is done by sending a notification to all clients subscribed to the control resource, using the subresource as key and the JSON keyword "null" as value.

The following example shows the SSE data notification for the removal of the resource /api/device/dsp/modules/agc/10

```
{
    "/api/device/dsp/modules/agc/10":  null
}
```

If the subresource has resources below it, the deletion notification MUST only be sent for the subresource. When the subresource is deleted, all resources below it are implicitly deleted.

Example for dynamic resources

The following example shows a pattern allowing for ordered creation, monitoring, updating and deletion of dynamic resources:

```
/api/device/dsp/modules/agc:
  get:
    summary: Returns list of instantiated AGC modules
    description: Returns list of instantiated AGC modules
    x-sennheiser-sscv2-resourcetype: [ "ControlResource" ]
    x-sennheiser-sscv2-subresource: "/api/device/dsp/modules/agc/{instanceId}"
    responses:
      '200':
        description: Successful request
        content:
          application/json:
            schema:
              type: array
              items:
                type: object
                properties:
                  instanceId:
                    type: string
                  value1:
                    type: string
                  value2:
                    type: string
                  value3:
                    type: string
      '403':
        description: Access permission not sufficient
  post:
    summary: Instantiate a new AGC module
    description: Instantiate a new AGC module
    requestBody:
      content:
        application/json:
          schema:
            type: object
            properties:
              value1:
                type: string
              value2:
                type: string
              value3:
                type: string
    responses:
      '201':
        headers:
          Location:
            description: Location of the newly created resource
            schema:
              type: string
        description: Successful creation
      '403':
        description: Access permission not sufficient

/api/device/dsp/modules/agc/{instanceId}:
  parameters:
    - name: instanceId
      in: path
      description: Instance to update
```

```
    required: true
    schema:
      type: string
get:
  summary: Return the currently configured values
  description: Return the currently configured values of the AGC module at instanceId
  responses:
    '200':
      description: Successful request
      content:
        application/json:
          schema:
            properties:
              value1:
                type: string
              value2:
                type: string
              value3:
                type: string
    '403':
      description: Access permission not sufficient
put:
  summary: Change AGC module at instanceId
  description: Change AGC module at instanceId
  requestBody:
    content:
      application/json:
        schema:
          type: object
          properties:
            value1:
              type: string
            value2:
              type: string
            value3:
              type: string
  responses:
    '200':
      description: Successful request
    '403':
      description: Access permission not sufficient
delete:
  summary: Delete and Deinstantiate the AGC module at instanceId
  responses:
    '200':
      description: Successful deletion
    '403':
      description: Access permission not sufficient
```

OpenApi vendor extensions

Special functionality of the SSCv2 API, which is exceeding the generic OpenAPI specification, is marked by the following OpenApi vendor extensions.

x-sennheiser-sscv2-resourcetype

This resource is used to signal special resources, which have behavior exceeding normal REST semantics. The following resource types are specified:

- FastResource: A resource, which is expected to update frequently. This allows a consumer of the Api to shape the traffic by only subscribing to this resource, when working with the data.
- ControlResource: A resource, which is following the special semantics laid out in Dynamic Resources. Subscribing to this resource will not trigger subscription updates for the ControlResource, but updates for every subresource instead.
- RebootingResource: A resource, which will trigger a reboot on PUT/POST interaction.

The resource type vendor extensions MUST be used on specific operations.

Please Note, that prerelease versions of the 2.3 specification standardized usage of a tag "FastResource". Consumers of SSCv2 api specifications MUST treat a tag "FastResource" identically to the respective x-sennheiser-sscv2-resourcetype vendor extension, if they encounter it, while creators of SSCv2 api specifications MUST NOT use the tag FastResource for new specifications.

x-sennheiser-sscv2-subresource

This vendor extension MUST only be used, when the resource is of the resource type ControlResource. It MUST contain the dynamic resource controlled by this control-resource and is used as a hint for code generation.

Sennheiser Sound Control Protocol v1

This protocol allows to command and to control the Sennheiser devices supporting the older SSC v1 version.

Below you will find a list of Sennheiser products that support the SSCv1 protocol. Click on the provided link to display the appropriate protocol for the desired Sennheiser product.

> **i** Note that the protocol is intended for command and control. Network audio streaming is entirely out of its scope.

SpeechLine Digital Wireless

SSC Protocol for SpeechLine Digital Wireless devices:

- SL Rack Receiver
- CHG 4N - network-enabled charger
- CHG 2N - 2 bay network charger
- Multi-channel receiver (SL MCR2 & MCR4)

Evolution Wireless Digital (firmware < 4.0.0)

SSC Protocol for Evolution Wireless Digital devices:

- EW-DX EM 2 rack receiver (EW-DX EM 2)
- EW-DX EM 2 rack receiver Dante (EW-DX EM 2 Dante)
- EW-DX EM 4 rack receiver Dante (EW-DX EM 4 Dante)
- CHG70N - 2 bay network charger

TeamConnect

SSC Protocol for TeamConnect devices:

- TeamConnect Ceiling 2 (TCC 2)

4. OpenAPI

Here you can find all product specific HTTPs methods, parameters and responses at a glance.

Currently the REST API is supported by:

- TeamConnect Bar S and M
- Team Connect Ceiling Mic Medium
- MobileConnect
- Evolution Wireless Digital (firmware ≥ 4.0.0)

> **i** Any future supported Sennheiser products will be added to this page. Please note that Sennheiser products that are not found on this page, support the Sennheiser Sound Control Protocol v1, and you can find their specifications on their respective here.

Evolution Wireless Digital (EW-DX)

EW-DX OpenAPI 1.7

All HTTPs methods, parameters and responses for EW-DX devices at a glance.

Supported EW-DX devices

- EW-DX EM2
- EW-DX EM2 Dante
- EW-DX EM4 Dante

> ⚠ Firefox Enhanced Tracking Protection (ETP), especially in “Strict” mode, can block scripts or cookies required by Swagger UI, causing it not to load, showing “Fetch error” messages, or breaking the “Try it out” function. To resolve this, disable Enhanced Tracking Protection for the specific Swagger page only:
>
> - Click the shield icon to the left of the URL in the Firefox address bar.
> - Turn off the **Enhanced Tracking Protection** toggle for this site.
> - Wait for the page to reload automatically and verify that Swagger UI works as expected.

> **i** The OpenAPI specification is only available online and can be accessed through the following link: Sennheiser 3rd party API

TeamConnect Bar (TC Bar S/M)

Known Issues

Here you can find all known issues regarding the methods, parameters and responses for TC Bar S/M.

General

Please note the following information:

- There is no **PUT** method for the request `/api/interfaces/network/dante` . You can find the **GET** method under **Device**.
- The **PUT** method for the request `/api/video/output/hdmi` (listed under **Video**) is out of function.
- The **PUT** request for enabling the camera `/api/video/input/internalCamera/enable` is deprecated.
- There is no PUT method available for the request `/api/video/input/internalCamera/videoParameters` .

Activating Bluetooth®

To activate the Bluetooth® function via the **PUT** request `/api/interfaces/bluetooth` , please only send the pairing request after 10 seconds. Requesting both values at the same time does not work.

- Send the first request with:

```
{
  "enabled": true
  }
```

- After 10 sec send the second request:

```
{
  "pairing": false
  }
```

TC Bar OpenAPI 1.16

All TeamConnect Bar HTTPs methods, parameters and responses at a glance.

Firefox Enhanced Tracking Protection (ETP), especially in “Strict” mode, can block scripts or cookies required by Swagger UI, causing it not to load, showing “Fetch error” messages, or breaking the “Try it out” function. To resolve this, disable Enhanced Tracking Protection for the specific Swagger page only:

- Click the shield icon to the left of the URL in the Firefox address bar.
- Turn off the **Enhanced Tracking Protection** toggle for this site.
- Wait for the page to reload automatically and verify that Swagger UI works as expected.

The OpenAPI specification is only available online and can be accessed through the following link: Sennheiser 3rd party API

TeamConnect Ceiling Medium (TCC M)

TCC M OpenAPI 1.9

All TeamConnect Ceiling Medium HTTPs methods, parameters and responses at a glance.



Firefox Enhanced Tracking Protection (ETP), especially in “Strict” mode, can block scripts or cookies required by Swagger UI, causing it not to load, showing “Fetch error” messages, or breaking the “Try it out” function. To resolve this, disable Enhanced Tracking Protection for the specific Swagger page only:

- Click the shield icon to the left of the URL in the Firefox address bar.
- Turn off the **Enhanced Tracking Protection** toggle for this site.
- Wait for the page to reload automatically and verify that Swagger UI works as expected.

The OpenAPI specification is only available online and can be accessed through the following link: Sennheiser 3rd party API

Subscription for real-time updates

This script sets up a subscription to receive real-time updates from a Sennheiser device.

The
script uses HTTP Basic Authentication to authenticate with the device and requests an event stream fro

Once the stream is established, it adds a subscription for a specific request and listens for incoming d
The incoming data is printed to the console as a string representation.
If the stream fails to establish the reason for the failure is printed to the console.

To set up a subscription for real-time updates:

- Replace the `ip` variable with the IP address of your Sennheiser device.
- Replace the `request` variable with the API request you want to subscribe to.
- Replace the `password` variable with the password for your Sennheiser Smart Control device.

```
Sennheiser electronic GmbH & Co. KG
Version: 1.0.0
Date: 2023-09-18
"""

# Import necessary libraries
from requests.auth import HTTPBasicAuth
import requests

# Set up variables for authentication and API requests
ip = '192.168.42.100'  # Replace with the IP address of your Sennheiser device
request = '/api/audio/inputs/microphone/beam/direction'

# Replace with the API request you want to subscribe to
password = 'the_future_of_audio'  # Replace with the password for your Sennheiser Sma
rt Control device

# Set up HTTP Basic Authentication and request an event stream from the device's API
auth = HTTPBasicAuth(
'api',
password
)
stream = requests.get(
f'https://{ip}:443/api/ssc/state/subscriptions',
auth=auth, stream=True,
headers={'Accept': 'text/event-stream'},
verify=False
)

# If the stream is established, add a subscription for the specified request and list
en for incoming data
if stream.status_code == 200:
stream.uuid = stream.headers['Content-Location'].split('/')[-1]
requests.put(
f'https://{ip}:443/api/ssc/state/subscriptions/{stream.uuid}/add',
json=[request], auth=auth,
verify=False
```

```
)

while True:
data = stream.raw._fp.fp.read1(1024)
data = data.decode('utf-8')
print(repr(data))
# If the stream fails to establish, print the reason for the failure to the console

else:
print(stream.reason)
```

Spectera

World´s first wideband bi-directional WMAS solution.

Please navigate to the desired chapter of the Spectera product documentation.

Introduction to Spectera API

This introduction shall assist as a starting point for integrators to be able to integrate and connect the Spectera system with their 3rd-party client applications.

Understand the Spectera system

We have a couple of documents and videos on YouTube available, which might help getting familiar with the system behavior and setup of Spectera compared to todays known single carrier systems.

- Spectera product website (Specs and Downloads)
- Spectera user manual and documentation
- Spectera Videos - YouTube Playlist

The communication between a 3rd-party client and the Spectera system will always be established via the Spectera Base Station network interface only. All other system components, like Antennas (DAD) or Mobile Devices (SEK), are aggregated dynamically via the API of the Base Station.

Understand the Sennheiser Sound Control (SSCv2)

A general introduction to our SSC protocol and the usage is provided on this website. Spectera does support Sennheiser Sound Control Protocol v2 **(v2.3)**, with a high security standard, only.

- Sennheiser Sound Control Protocol (v2.3) - Specification

Spectera Security Guide

A general introduction to all the security features of the Spectera system is documented in the Spectera Security Guide. This guide also explains in detail how to uses the device authentication incl. reset of device password.

- Spectera Security Guide



Spectera API supports encrypted HTTPS communication on port 443 only. HTTP requests on default port 80 or port 443 will be rejected.

Authentication

It is mandatory to authenticate on the device with each HTTPS request.



Since the 3rd-party "API" access level is not implemented yet on Spectera, you have to use the existing access level used by our Sennheiser applications and NOT the "API" access level mentioned inside the general SSCv2 documentation.

- Using HTTP basic authentication
- **Username**: controlSennheiser
- **Password**: configured using Sennheiser LinkDesk or WebUI

Brute-force prevention

Spectera Base Station has implemented a prevention mechanism making brute-force attacks on network interfaces impracticable. This ensures that whenever a client with a dedicated IP address tries to access a resource with wrong/invalid credentials repeatedly, this client with the dedicated IP address is blocked for a given time. The device does not keep the state for brute-force prevention across reboots.

Understand the Spectera API

This website publishes a YAML file with the OpenAPI specification of the Spectera API with all ressources accessable by the "API" access level.

Interface Versioning

At Sennheiser we follow a policy for versioning of the API. Here an abstract:

Introduction

Devices with an SSCv2 interface can be controlled and managed by clients over a REST-API. The functionality of the devices evolves over the time with new versions of the API, which add new features and modifies existing behavior.

This policy shall provide some rules and guidelines, how the versioning of an SSCv2 REST-API shall be treated.

Brief outline on Semantic Versioning

This document refers to Semantic Versioning Specification. Here an adapted short summary from the specification.

> **i** Given a version number MAJOR.MINOR.PATCH, increment the:

- MAJOR version when you make incompatible API changes
- MINOR version when you add functionality in a backward compatible manner
- PATCH version when you make backward compatible bug fixes

Additional labels for pre-release and build metadata are available as extensions to the MAJOR.MINOR.PATCH format.

A pre-release version MAY be denoted by appending a hyphen and a series of dot separated identifiers immediately following the MINOR version. Identifiers MUST comprise only ASCII alphanumerics and hyphens [0-9A-Za-z-]. Identifiers MUST NOT be empty. Numeric identifiers MUST NOT include leading zeroes. Pre-release versions have a lower precedence than the associated normal version. A pre-release version indicates that the version is unstable and might not satisfy the intended compatibility requirements as denoted by its associated normal version. e.g. 1.0-alpha

Source: semver.org

Technical contract

Public SSCv2 Interface Version (client readable)

The SSCv2 Specification already defines a path and schema, that a client system is able to to retrieve the version information from the SCCv2 API of a device. This includes the protocol version and the schema version of the API.

```
/api/ssc/version:
  get:
    summary: Get the versions relevant for SSCv2 implemented
    description: An SSCv2 Command to retrieve the SSCv2 protocol and schema versions
 implemented in the SSCv2 Server
```

```
version:
    type: object
    properties:
        protocol:
            type: string
            example: "2.3"
            description: The version of SSC protocol.
        schema:
            type: string
            example: "17.0"
            description: This is the schema version of the API of the device. Semantic
 versioning must be used. So additional paths or properties will increase the minor
 number while breaking changes increase the integer part before the decimal point.
```

The schema version MUST follow the Semantic Versioning Specification.

⚠

- For protocol and schema versions use only the MAJOR.MINOR parts of the version string and do not use the PATCH part nor any other additional label as prefix or suffix.

- Both properties must be machine-readable and interpretable as numbers in order to perform numerical comparisons. The PATCH level is used if only the documentation of the YAML file has changed, e.g. the "description" property, which has no effect on the data transferred at the interface. Therefore, a client does not need to receive this information.

OpenAPI Specification Version

For the definition and documentation of an SSCv2 REST-API we are using OpenAPI Specification which already defines objects including the version information for the OpenAPI Specification being used and the version of the YAML file itself.

```yaml
openapi: 3.1.0
info:
  title: SPECTERA API
  version: v17.0.1
```

Breaking Changes, Deprecated and Preliminary Features

> ⚠ Upcoming API versions may introduce breaking changes (reflected by the Major number of the API version string).

Sennheiser is trying to avoid breaking changes with new API versions as long as possible. Over the time we might need to refactor or improve functionality at the API. If we are going to change or even remove API functionality, we will give a grace period by deprecating the functionality officially. After an announced sunset date the deprecated functionality will be removed with a new API version.

In the OpenAPI specification the affected object will be marked with the `deprecated` property.

```
Connected:
  type: boolean
  description: |
    **DEPRECATED**: This property is going to be removed with the next major version of
 the API
    - Use the `state` property with values "Connected" and "Disconnected" instead.
    - The value is "true" if the state equals "Connected" or "Updating", otherwise the
 value is "false".
  deprecated: true
```

Similarly it is about new functionality, which get's introduced by new API versions. Sometimes the functionality is not fully implemented or tested yet, but is already visible at the API. Such affected object will be just marked within the `description` property and will be mentioned in the release notes.

```
Sleep:
  type: boolean
  description: |
    **PRELIMINARY**: The Sleep Mode is a preliminary feature. The functionality is not
 fully implemented yet and the behavior may change in future versions of the API.
```

Usage of Spectera API

Enabling API Access

Sennheiser devices cannot be accessed via the API in factory default state. In order to enable it, you have connect to the Base Station via Sennheiser LinkDesk or Spectera WebUI and set a password. The Spectera Security Guide explains in detail how to set a device password.

Connecting to the Base Station

Use the IP address of the device and the HTTPS port 443 to form the base URL for the connection requests, e.g. "https://192.168.0.1:443".



Since the 3rd-party "API" access level is not implemented yet on Spectera, you have to use the existing access level used by our Sennheiser applications and NOT the "API" access level mentioned inside the general SSCv2 documentation.

- Using HTTP basic authentication
- **Username**: controlSennheiser
- **Password**: configured using Sennheiser LinkDesk or WebUI

Communicating with the API

SSCv2 uses two modes of message exchange:

- Synchronous: The client sends a GET/PUT/POST/DELETE request and the server immediately sends a response
- Asynchronous: The client subscribes to parameter changes and the server notifies the client of parameter changes via a Server Sent Event (SSE). For more see Sennheiser Sound Control Protocol (v2.3) - Specification.

Please also have following **Spectera specific recommendations** in mind:

> **i** Use subscriptions for update notifications
>
> - We always recommend to use the subscription mechanism to receive updates when resources/objects have been changed instead of polling the resource/object with the GET method.
> - The SSC server on the Base Station has a limit on how many connections can be handled concurrently. For the sake of fairness to other potential clients, try to keep the number of parallel open connections as less as possible. E.g. keep just one connection for asynchronous (subscriptions) and another for synchronous communication open or open them just on demand.

> **i** Coordinating Multi-Client-Write-Access
>
> - Currently the Multi-Client-Write-Access is not fully implemented and tested yet and might have side effects if more than one client is writing to a resource at the almost same time. Therefore the `usagemark` property comes into place. `/api/ssc/usagemark` .
> - There is no active lock of write access, but the `usagemark` can be used to inform other clients, that they are connected with the Base Station and doing write requests. Other clients shall switch actively into "read-only" access, if the `usagemark` is or is getting set by another client. By setting the `usagemark` property a timeout has to be applied, when the `usagemark` get's reset automatically by the Base Station.
> - **Please be fair with the `usagemark` and do not hold it longer than necessary.**
> - More details are below in example section and the YAML spec.

Managing Mobile Devices

> **i** Don't get confused with the abbreviation "MT" or "MTs" in the API paths or in general. It's a synonym for "Mobile Device".

Each Mobile Device (SEK, SKM, etc.) which is connected to the Base Station is handled as a dynamic resource and can be accessed via a unique URI (Uniform Resource Identifier). A group of multiple devices are handled with collections or lists of resources. Each collection can be managed with the typical REST API methods `GET/PUT/POST/DELETE`.

There are 2 collections for handling the Mobile Devices depending on their current state of relation to the Base Station.

List	API path	Description
Pairable Devices	`/api/mts/pairable`	Contains all the objects representing the Mobile Devices which are currently connected to the Base Station (in Pairing Mode) and can potentially get paired
Paired Devices	`/api/mts/paired/all`	Contains all the objects representing the Mobile Devices which are currently paired to the Base Station regardless of whether they are currently connected or disconnected

Specific devices can be accessed by `GET` and `PUT` methods on the dedicated resource containing the specific `mtUid`, e.g. `/api/mts/pairable/{mtUid}` or `/api/mts/paired/all/{mtUid}`

mtUid vs. serial number

The **mtUid** is an unique identifier for each Mobile Device inside the Spectera system regardless the type of device. At the API the property `mtUid` is used to address a specific Mobile Device resource.

The **serial number** is an unique identifier for each type of Mobile Device. That means that different types of Mobile Device can potentially have the very same serial number. Therefore it can not be used as unique identifier to address a specific Mobile Device resource at the API. Nevertheless can it be read with the property `serial` of an Mobile Device resource. The serial number is also printed at the housing of a device.

Pairing a Mobile Device

Before configuring Audio Links on a specific Mobile Device it needs to be paired to the Base Station. This can be done by using the Spectera WebUI (Pairing/unpairing mobile devices) or by using the API.

Basic steps are:

1. Activate the Pairing Mode of the Base Station - `PUT /api/mts/pairing` with `enable:true`
2. Set the Mobile Device into Pairing Mode - Switching the SEK on and off
3. Check the list of Pairable Devices - `GET /api/mts/pairable`
4. Verify/Compare the `PairingPinCode` of the specific Mobile Device going to be paired - `GET /api/mts/pairable/{mtUid}`
5. Add the Mobile Device to the list of Paired Devices - `POST /api/mts/paired/all` with the `mtUid`
6. The Mobile Device reconnects and will be available in the list of Paired Devices - `GET /api/mts/paired/all`
7. Deactivate the Pairing Mode of the Base Station - `PUT /api/mts/pairing` with `enable:false`

A Mobile Device in the list of Pairable Devices can be forced to search for another Base Station to connect to by removing it from the list. `DELETE /api/mts/pairable` with the `mtUid`

Unpairing a Mobile Device

A Mobile Device can be actively un-paired from the Base Station by removing it from the list of Paired Devices. `DELETE /api/mts/paired/all/{mtUid}`

Managing Audio Links

About Audio Links

Audio Links are dynamic and virtual resources allocating timeslots in the RF channel. The number of allocated timeslots depends on the "mode" of the Audio Link. Modes with low latency, for example, need more timeslots. Creating an Audio Link means reserving the required amount of timeslots. The create fails if not enough timeslots are available. The table below lists the available modes and their size in number of required timeslots. One RF channel has a capacity of 128 timeslots.

The full table of available Audio Link Modes and their characteristics can be found in the Spectera System Specification.

The headlines of the short table below are following:

- Name: Name of the Audio Link Mode
- Ch/Link: Audio channels per Audio Link, where 1 means mono and 2 means stereo
- ID: Audio Link Mode ID used for configuring the Audio Link resource
- IEM: Available for IEM links
- MIC: Available for MIC links
- Slots: Amount of slots that are occupied out of the 128 slots available per RF channel

Name	Ch/Link	ID	IEM	MIC	Slots
Deactivated		0			
`LIVE Ultra Low Latency`	2 (Stereo)	9	✓	-	32
`LIVE Low Latency`	2 (Stereo)	8	✓	-	16
`LIVE`	2 (Stereo)	7	✓	-	8
`LIVE Link Density`	2 (Stereo)	6	✓	-	4
`RAW Low Latency`	1 (Mono)	11	-	✓	16
`RAW`	1 (Mono)	10	-	✓	8
`LIVE Low Latency`	1 (Mono)	5	-	✓	16
`LIVE`	1 (Mono)	4	✓	✓	8
`LIVE Link Density`	1 (Mono)	3	✓	✓	4
`MAX Range`	1 (Mono)	1	✓	✓	8
`MAX Link Density`	1 (Mono)	2	✓	✓	1
`Empty (Mono)`	1 (Mono)	1001	✓	✓	0

Name	Ch/Link	ID	IEM	MIC	Slots
`Empty (Stereo)`	2 (Stereo)	1002	✓	-	0

> **i** The `Empty` Audio Link Modes can be used to already define an audio routing between an Audio Source and Sink without allocating timeslots yet. Therefore no audio will be transmitted.

Configuration

The audio signal routing between Mobile Devices and Base Station input and output channels can be configured flexible by creating, modifying and removing Audio Links and assigning an existing Audio Link to a Source and a Sink (either Mobile Device or Audio I/O). The list of Audio Links can be managed with the typical REST API methods `GET/PUT/POST/DELETE` on the API path `/api/audio/links`.

Creating a signal routing always consist of three basic steps:

1. Create an Audio Link resource by defining the desired Audio Link Mode and which RF channel to be used - `POST /api/audio/links`
2. Assign the Audio Link resource to the desired Mobile Device - `PUT /api/mts/paired/all/{mtUid}`
3. Assign the Audio Link resource to either an input or output channel of the Base Station - `PUT /api/audio/inputs/{inputId}` for an IEM link or `PUT /api/audio/outputs/{outputId}` for a MIC link

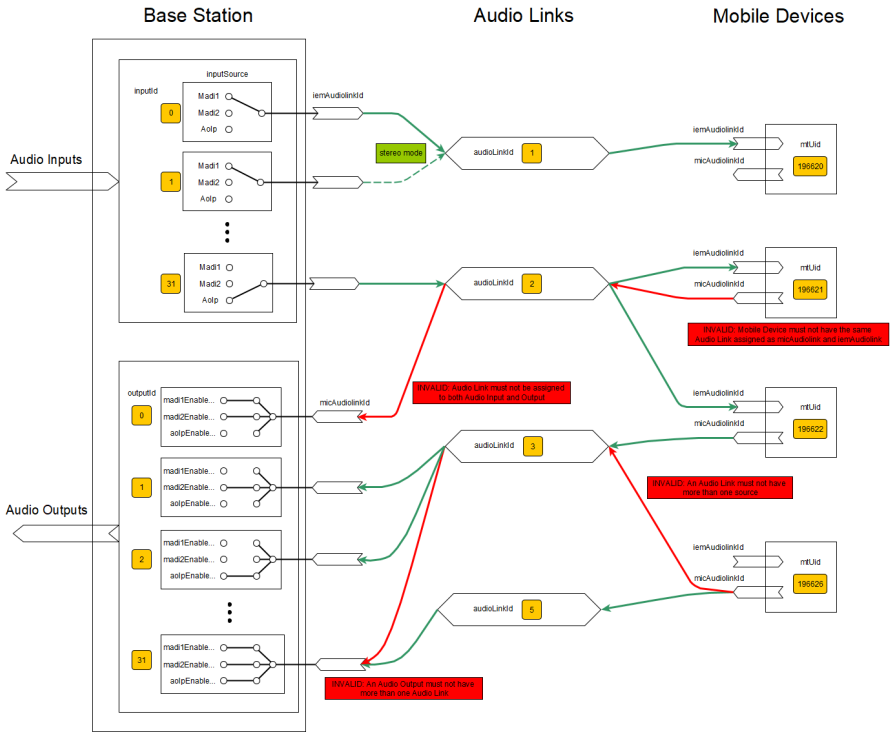


Please be aware that inconsistent Audio Link resources can be removed by the "Garbage Collection" of a running Spectera WebUI instance. Please ensure that steps 1 and 2 are performed consecutively within 5 seconds to avoid removal of the Audio Link resource. More information can be found on the section "Garbage Collection" below.

Validation Rules

Below rules apply when dealing with Audio Links, e.g. creating, updating, as well as when assigning to other resources as Mobile Devices and Base Station in/outputs:

- An Audio Link must not have more than one source
- Audio Link and Mobile Device must be in the same RF-Channel
- The Audio Links must not exceed the channel capacity (number of slots ≤ 128)
- An Mobile Device cannot have the same Audio Link assigned as micAudiolink and iemAudiolink
- Some Audio Link Modes are pure MIC/IEM modes
- Stereo links are assigned to even input indices, the next odd index is implicitly assigned to the same link
- An Audio Link must not be assigned to both input and output
- The output interfaces must not all be off.
- An Mobile Device cannot change its RfChannel if it has an Audio Link assigned to
- An Audio Link cannot be changed between MONO and STEREO mode


Base Station
Audio Links
Mobile Devices
Audio Inputs
Audio Outputs
inputSource
inputId
Madi1
Madi2
Aoip
iemAudioLinkId
micAudioLinkId
stereo mode
audioLinkId
mtUid
196620
196621
196622
196626
outputId
madi1Enable...
madi2Enable...
aoipEnable...
INVALID: Audio Link must not be assigned to both Audio Input and Output
INVALID: Mobile Device must not have the same Audio Link assigned as micAudioLink and iemAudioLink
INVALID: An Audio Link must not have more than one source
INVALID: An Audio Output must not have more than one Audio Link

Pending Actions

The API of the Base Station is always buffering all incoming Audio Link configuration requests. Still, applying these changes may take some time due to internal processing delays.

The status of Pending Actions can always be accessed on the API path `/api/audio/pendingactions`.

It is up to the API client implementation to stop pushing further requests or waiting for the pending actions to complete.

Garbage Collection - Removing "orphan" Audio Links

The current design of the API allows (by inconsistent configuration) to have Audio Link resources allocating timeslots although they are not assigned to any source or sink, so-called 'orphan links'. If any Spectera WebUI instance running on a client browser has access to the Base Station, it regularly checks if such 'orphan links' exist and starts to clean it up by removing this resources. An Audio Link resource needs to have at least a Mobile Device assigned, to not get removed.



Please be aware that a connected Spectera WebUI instance can remove 'orphan links' at any time. Still the WebUI is respecting the `usagemark` property. `/api/ssc/usagemark`

Command Mode

> **Command Mode** feature has been officially introduced with the Spectera API version 18.1. In previous API versions, this feature is not officially supported. Please check the Supported API versions for more details on available firmware versions.

About Command Mode

The feature **Command Mode** provides the possibility for the user of a Mobile Device to trigger a custom routing of the audio signal on the Base Station by pressing a dedicated Command Button at the Mobile Device.

> The custom routing in Command Mode is only available for MIC Audio Links respectively Audio Outputs at the Base Station and not for IEM Audio Links and Audio Inputs.

Following functionality can be configured:

- The Command Mode can be enabled for each Mobile Device separately.
- The Command Button behavior can be configured to be either "Momentary" or "Latching". In "Momentary" mode the button needs to be pressed and held to activate the command, in "Latching" mode one press activates the command and another press deactivates it.
- For Command Buttons with LED indication (e.g. SKM COM) the color can be customized for all the available Command Mode states.
- For each Audio Output at the Base Station the routing to the Audio Interface (AoIP, MADI1, MADI2) can be configured individually for the Command Mode being enabled or not. This allows to have a custom audio routing when the Command Mode is active, e.g. for a talkback function.
- The audio routing behavior can be configured in 2 styles:
    - Mute: The audio signal is muted when the Command is triggered and unmuted if not
    - Talk: The audio signal is unmuted when the Command is triggered and muted if not
- Since an Audio Link can be assigned to several Audio Outputs, the Command Mode can be used for custom audio routing of one Mobile Device to different Audio Outputs, depending on the Command trigger.

Additionally the state of the Command Button can be accessed via the API and used for triggering other actions on the Base Station or at the client side, e.g. for controlling external devices or for visualizing the state of the Command Button.

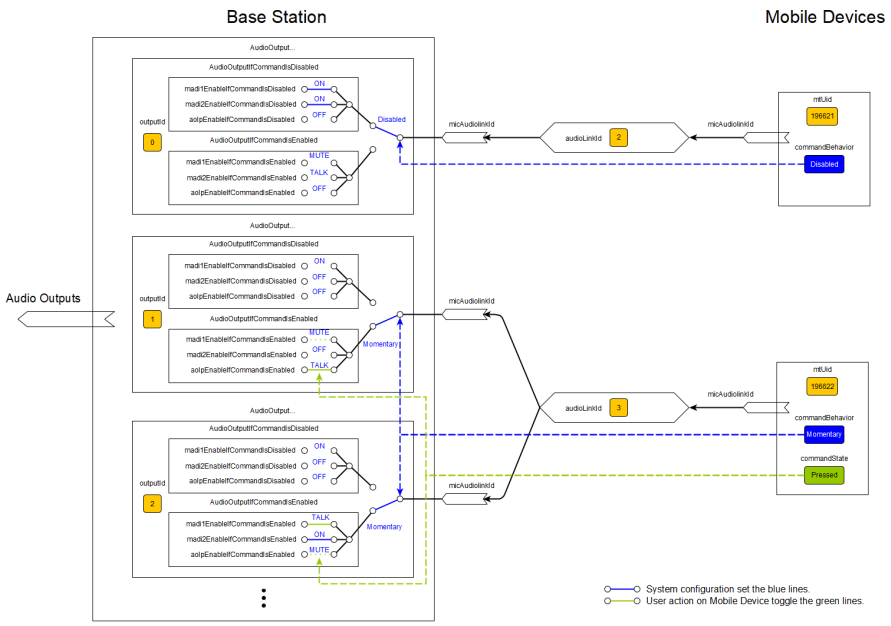
Configuration

The Command Mode behavior and audio routing can be configured by modifying the properties of a paired Mobile Device and of the Audio Outputs at the Base Station.

Configuring the Command Mode always consist of two basic steps:

1. Enable the Command Mode for the desired Mobile Device by defining the behavior (Momentary or Latching) at the property `commandBehavior` - `PUT /api/mts/paired/all/{mtUid}`
2. Configure the state dependent audio routing (`On` , `Off` , `Mute` , `Talk`) for the Command Mode at the desired Audio Outputs by defining the desired routing at the properties `aoIpEnableIfCommandIsEnabled` , `madi1EnableIfCommandIsEnabled` , and `madi2EnableIfCommandIsEnabled` - `PUT /api/audio/outputs/{outputId}`

> **i**
>
> - The interfaces of an Audio Output with an Audio Link assigned must not all be `Off` .
> - Even with Command Mode enabled the routing can be set to `On` or `Off` . In this case the Command trigger will not change the routing, but the state of the Command Button can still be accessed via the API and used for other purposes.



Retrieve Live Monitoring Data with Level Recording

> **i** **Level Recording** has been introduced with the Spectera API version 18.1 as preliminary feature which will continuously evolve. It is not available in previous API versions. Please check the Supported API versions for more details on available firmware versions.

About Level Recording

The feature **Level Recording** provides a stream of telemetry data, e.g. audio metering, Mobile Device parameters, antenna and RF channel parameters, which can be used for monitoring and debugging purposes. A certain set of parameters, which can be accessed via the SSCv2 API, are available through Level Recording but are provided as a high-efficient binary stream encoded with FlatBuffers instead of JSON. The Level Recording API is designed as a sub-schema of the SSCv2 Control API and therefore follows the same authentication and subscription mechanism as the main SSCv2 OpenAPI. The only difference is that the payload of the SSE events is encoded as FlatBuffers and not JSON.

The Base Station is generating the Level Recording data as snapshots in regular intervals of around 1 second and provides it via a dedicated endpoint as SSE events with FlatBuffers-encoded payload (Frame). Each Frame is structured in different "sections" for better maintainability. Each section contains a set of parameters related to a specific topic, e.g. audio metering, etc. The sections are defined in a dedicated FlatBuffers schema (*.fbs file). Some of these sections contain bulk data of the 1 second snapshot, e.g. the audio metering data provides 64 values per audio channel for each Frame.

Additionally to the live stream of monitoring data, the Base Station also provides the possibility to retrieve historical data of the last 1 hour with a time resolution of around 1 second. This option will mainly used to cover lost live data or to analyze past events. The historical data is provided with the same structure and schema as the live data, but only on demand when requested by the client. This can be achieved by providing the desired range (start and end index) of historical data to the dedicated endpoint for historical data retrieval. The Base Station will then send all available Frames of the requested range as SSE events with FlatBuffers-encoded payload.

Brief outline on FlatBuffers

FlatBuffers is a serialization library and format developed by Google. It allows to define data structures in a schema and serialize them into a compact binary format. The main advantage of FlatBuffers is that it allows zero-copy deserialization, which means that the data can be accessed directly from the serialized buffer without the need for parsing or unpacking, making it very efficient for real-time applications. The schema of the FlatBuffers data is defined in a *.fbs file, which can be compiled to generate code for different programming languages to easily encode and decode the data.

Schema Versioning

The FlatBuffers schema has its own life-cycle and therefore versioning independent of the SSCv2 API versioning. A new version of the FlatBuffers schema will be introduced and announced with at least a change of the minor SSCv2 API version, but not necessarily with every new API version.

Sennheiser follows the official FlatBuffers schema evolution guidelines with the primary goal of wire compatibility of the binary payload across minor schema versions:

> **i**
>
> - **Wire compatibility across minor schema versions:** The binary format remains decodable across minor schema version changes (e.g. 1.0 → 1.1). Newer clients can decode older payloads and vice versa.
> - **Feature compatibility is not guaranteed across minor schema versions:** While the binary format stays decodable, individual fields may be deprecated (returning default/zero values). Features depending on deprecated fields will no longer receive meaningful data. Refer to the release notes for details on deprecated and newly introduced fields.
> - **Major schema versions may break wire compatibility:** A major schema version increment (e.g. 1.x → 2.0) may require client-side decoder updates.

Since the FlatBuffers schema covers a similar set of parameters as the SSCv2 API, the schema evolution will be primarily driven by the changes in the SSCv2 API schema. Deprecated parameters and breaking changes in the FlatBuffers schema will be announced in the release notes.

> **i** Level Recording clients should use the schema version reported by `/api/levelrecording/version` as their version contract and check it at connection time to detect schema changes independently of the SSCv2 API version. As the Level Recording feature is still preliminary, the versioning and evolution policy may be refined in future API versions.

Endpoints overview

GET /api/levelrecording/version

- Get the Level Recording schema version

Example Response Body:

```
{
  "protocol": "flatbuffers",
  "schema": "1.0",
  "schemaDetailed": "1.0.0"
}
```

The version information is relevant for decoding the FlatBuffers payload of the Level Recording data. The client needs to ensure that the schema version is compatible with the version it supports to be able to decode the payload correctly.

GET /api/levelrecording/info

- Get information about currently available indices and buffer fill level of the historical level recording data.

Example Response Body:

```
{
  "session_uuid": "42b3f67c-9ee5-438b-ac92-00fa71e040e2",
  "start_index": 0,
  "size": 148
}
```

This information is relevant for retrieving historical data with the endpoint `/api/levelrecording/history` as it provides the valid range of indices for the historical data and the session UUID to be used for retrieving the historical data. The session UUID is unique for each Level Recording session and can be used by the client e.g. after a reconnection to check if the historical data has been reset or if it is still available for retrieval. Every restart or reset of the Base Station will reset the Level Recording session and therefore all historical data will be lost and a new session UUID will be generated.

GET /api/levelrecording/live

- Request Level Recording Live Data by initiating a subscription to the live stream of Level Recording data as Server Sent Events (SSE) with FlatBuffers-encoded payload.

This GET request will not return a single response body, but initiates a subscription to the live stream of Level Recording data as Server Sent Events (SSE). The client will receive an SSE event every second with the latest snapshot of monitoring data encoded in the payload as FlatBuffers binary data according to the Level Recording schema. The client needs to decode the FlatBuffers payload according to the schema to access the monitoring data.

Following SSE event types are available in the live stream:

event:open\ndata:<json-data>\n\n

- This event is sent once when the subscription to the live stream is established. The data payload contains a JSON object with the session UUID, the start index for the first FlatBuffers Frame, the current buffer fill level of the historical data, and the version of the FlatBuffers schema (schemaDetailed).

Example Payload:


```
{
  "session_uuid": "42b3f67c-9ee5-438b-ac92-00fa71e040e2",
  "start_index": 0,
  "size": 148,
  "version": "v1.0.0"
}
```


event:message\ndata:<json-data>\n\n

- This event is sent for each new snapshot of Level Recording data. The data payload contains a JSON object with the index, timestamp, and the FlatBuffers-encoded data.

Example Payload:


```
{
  "index": 149,
  "time_stamp": 1770207951507,
  "payload": "GAAAAAAAEgAga..."
}
```


event:close\ndata:"goodbye"\n\n

- This event is sent when the subscription to the live stream is closed from server-side for any reason.

- The Level Recording data is only available as SSE stream. Therefore the connection needs to be kept open. There is no possibility to retrieve the data with single synchronous GET requests.
- We recommend to use a dedicated connection/socket for the live stream and not share it with other requests.
- The stream can be stopped by closing the connection.

This endpoint can not be subscribed with the usual SSCv2 subscription mechanism via `/api/ssc/state/subscriptions` .

GET /api/levelrecording/history

- Get historical Level Recording data by providing the start and end indices of the data to be retrieved.

This GET request will not return a single response body, but initiates a stream of historical Level Recording data as Server Sent Events (SSE). The client will receive SSE events as fast as the server can provide them. The client needs to decode the FlatBuffers payload according to the schema to access the monitoring data.

The SSE event types are the same as for the live stream, but the payload contains only the historical data of the requested range of indices.

i

- As with the live stream, the historical Level Recording data is only available as SSE stream.
- If the requested range of historical data is not available anymore, e.g. due to a session reset or because the requested indices are out of range, the server will send at least all the data still available within the requested range.
- We recommend to use a dedicated connection/socket for the historical stream and not share it with other requests.
- The stream can be stopped by closing the connection.
- The stream will be automatically stopped by the server (with a close event) after sending the requested range of historical data.



This endpoint can not be subscribed with the usual SSCv2 subscription mechanism via `/api/ssc/state/subscriptions` .

Examples of basic requests for 3rd-party access

Here are a few examples which resources might be useful for the 3rd-party access.

GET /api/ssc/version

- Get the SSCv2 protocol and schema (interface) versions

Example Response Body:

```
{
  "protocol":"2.3",
  "schema":"17.0",
  "schemaDetailed":"v17.0.0+0-g7081b87f"
}
```

GET /api/ssc/usagemark

- Check if the device is "in use"

Example Response Body, if the device is "in use":

```
{
  "usedBy": "LinkDesk.JaneDoe-12345abcdef",
  "timer": 5
}
```

Example Response Body, if the device is NOT "in use":

```
{
  "usedBy":"",
  "timer":300
}
```

PUT /api/ssc/usagemark

- Set the device as "in use" for the next 10 seconds.

Request body:

```
{
  "usedBy": "3rd-party.JoeDoe-12345abcdef",
  "timer": 10
}
```

GET /api/device/identity

- Get the device identity

Example Response Body:

```
{
  "product":"Spectera-Base-Station",
  "hardwareRevision":"0594192-08",
  "serial":"1424000001",
  "vendor":"Sennheiser electronic SE & Co. KG"
}
```

PUT /api/device/identification

- Sets the state of device identification. Let the LED of Base Station flash for identification.

Request body:

```
{
  "visual": true
}
```

GET /api/firmware/update/state

- Get the state of a firmware update process incl. version. For 3rd-party only the firmware version of the Base Station and of the Dante module might be of interest.

Example Response Body:

```
{
  "deviceVersion":"v1.3.0",
  "aoIpVersion":"1.1.0",
  "state":"Idle",
  "progress":0,
  "lastStatus":"None",
  "lastStatusDetailed":"NONE"
}
```

GET /api/rf/channels

- Get the list of RF channels and their settings.

Example Response Body:

```
[
  {
    "rfChannelId":0,
    "txPower":50,
    "frequency":522000,
    "bandwidthMode":8000,
    "rfRestrictionViolation":false,
    "rfState":"RfActive",
    "rfStateOnStartup":"RfMuted"
  },
  {
    "rfChannelId":1,
    "txPower":50,
    "frequency":554000,
    "bandwidthMode":8000,
    "rfRestrictionViolation":false,
    "rfState":"RfActive",
    "rfStateOnStartup":"RfMuted"
  }
]
```

GET /api/mts/paired/all

- Get the list of all paired Mobile Devices and all of their properties.

You will get an array of all Mobile Devices paired with the Base Station. Each entry contains all the properties of each single device. You can use the "mtUid" for further requests on a dedicated Mobile Device.

Example Response Body:


```
[
  {
    "mtUid":16789965,
    "type":"SEK",
    "frequencyRange":"UHF",
    "rfChannelId":1,
    "identify":false,
    "reverseIdentify":false,
    "name":"SEK",
    "serial":"1495000001",
    "sleep":false,
    "state":"Connected",
    "lastConnected":"NotAvailable",
    "version":"v1.3.5",
    "versionMismatch":false,
    "fccId":"DMOSEKUHF",
    "batteryFillLevel":100,
    "batteryRuntime":65535,
    "batteryLow":false,
    "ledBrightness":"Standard",
    "connectedStateColor":"#00A100",
    "swUpdatePossible":true,
    "swUpdateNotPossibleReason":"",
    "swUpdateProgress":-1,
    "micAudiolinkId":39,
    "iemAudiolinkId":-1,
    "micAudiolinkActive":true,
    "iemAudiolinkActive":false,
    "headphonePlugState":"Unplugged",
    "headphoneVolume":-20,
    "headphoneVolumeMax":27.5,
    "headphoneVolumeMin":-100,
    "headphoneBalance":0,
    "headphoneMonoBalanced":false,
    "micPreampGain":12,
    "micLowCutHz":20,
    "micTestToneEnabled":false,
    "micTestToneLevel":-60,
    "micLineSelection":"Auto",
    "micLineSelectionAutoValue":"Mic",
    "cableEmulation":"Off",
    "commandState":"NotAvailable",
    "iemLqi":0,
    "micLqi":4,
    "interference":
    {
      "severity":"Low"
    },
    "dominantAntenna":"d",
    "rssi":-42
  }
]
```

PUT /api/mts/paired/all/{mtUid}

- Modify the properties of a paired Mobile Device given by the "mtUid" and "type", e.g. let the LED flash to identify.

Example Request Body:

```
{
  "mtUid":16789965,
  "type":"SEK",
  "identify":true
}
```

GET /api/audio/levels

- Get the audio input (IEM) and audio output (MIC) levels at the Base Station.

You will get the audio input and output levels of all audio channels for all interfaces (Madi1, Madi2 and Dante).

> **i** We recommend to subscribe for that resource than pulling it with a GET request. You will receive always the full set of audio levels for all 32 input and 32 output channels on the Base Station. There is no dedicated resource or property to get the audio level of one dedicated Mobile Device. If you would like to know the mapping of a Mobile Device to an Audio Channel you have to do "reverse engineering" by reading out the Audio Links configured for a dedicated Mobile Device.

Spectera OpenAPI specification versions

Supported API versions

Each release of the Spectera Base Station supports a dedicated API version.

Spectera API version	LR sub-schema version	Spectera Base Station firmware version
18.3	1.1	≥ 1.5.0
18.1	1.0	≥ 1.4.0 up to < 1.5.0
17.0	-	≥ 1.3.0 up to < 1.4.0

⚠ Upcoming firmware releases may introduce new API versions and even breaking changes (reflected by the Major number of the API version string).

Spectera OpenAPI version 18.3

Release Notes Version 18.3

Latest information on the Spectera API, including detailed descriptions of new features, breaking changes, upcoming deprecations and known issues.

Version 18.3.0

New Features

- No new features have been added in this version

Improvements

- Command Mode: For command buttons with LED indication (e.g. SKM COM) the color can be customized now for all the available Command Mode states.
    - added property `commandColors` at `/api/mts/paired/all/{mtUid}` to configure the colors for the command button.
- Level Recording: New flatbuffers schema version 1.1 with a bunch of new properties for extended telemetry functionality and improved versioning and compatibility checks.
    - added `schema_version` property explicitly to the schema definition
    - added a bunch of new properties
        - `base_station` and `audio_interfaces` represent in their sub-properties the current states and configuration of the base station and audio interfaces at the time of the level recording
    - extended the `mic_link` property with `command_state` for Command Mode representation
    - extended the `seks` and `skms` property array with `command_behavior` for Command Mode representation
    - Please refer to the updated schema definition file for the complete list of new properties and their description.
- Audio Links: The request body for the POST request to `/api/audio/links` has been extended by the new property `linkedMt` to allow the atomic creation of audio links and assignment to an existing Mobile Device in one request. This ensures that Audio Links are always in use by a Mobile Device and allows for improved garbage collection of orphaned Audio Links. The new property is optional for being backwards compatible, but will become required in upcoming versions of the API.
- RF: Removed the fixed default values from the properties `frequency` and `bandwidthMode` at `/api/rf/channels/{rfChannelId}`. The default values are now set by the system based on the license entitlements and the current RF restrictions. Please read the current RF restrictions at `/api/rf/restrictions` to find out which frequencies and bandwidths are allowed for your system.
- Device: The request body for the GET request to `/api/device/identity` has been extended by the new property `variant` with its sub-properties `articleNumber`

and `articleName` to allow clients to differentiate between hardware variants of the Spectera Base Station.
- Network: The request body for the GET request to `/api/interfaces/network` has been extended by some new `one-of` options representing the new AoIP interface type (Ravenna) for Spectera Base Station ZMAN hardware variant.

> ⚠ Clients accessing Spectera Base Station ZMAN hardware variant need to support at least API version 18.3.

Breaking Changes

- No breaking changes have been introduced in this version

Upcoming Deprecations

> **i** These functionalities are still available in this version but will be removed in the next major version. Please update your implementation to use the new alternatives as soon as possible to avoid disruptions when upgrading to the next major version.

- Audio Links: The `linkedMt` property for the POST request to `/api/audio/links` will become required in the next major version of the API. Please ensure that all Audio Links are assigned to a Mobile Device when creating them.
- Audio Interfaces + Audio IO: As preparation for upcoming new audio interface the name of the interface `Dante` and `AudioNetwork` will be replaced with the more generic name `AoIP` (Audio over IP). The following endpoints and properties will be replaced by adding new and removing the old ones. The old endpoints and properties are marked as deprecated until they are removed with the next major version:
    - Audio Interfaces: Endpoints `/api/audio/interface/audionetwork/status` and `/api/audio/interface/audionetwork/clock/input/status` will be replaced by `/api/audio/interface/aoip/status` and `/api/audio/interface/aoip/clock/input/status`
    - Audio Interfaces: The `outputStatus/clockSource` property for the following endpoints will be replaced by `outputStatus/outputClockSource`
        - `/api/audio/interface/madi1/status`
        - `/api/audio/interface/madi2/status`
        - `/api/audio/interface/wordclock/status`
    - Audio Interfaces: The `clockSource` property for the following endpoints will be replaced by `outputClockSource`
        - `/api/audio/interface/madi1/output`
        - `/api/audio/interface/madi2/output`
        - `/api/audio/interface/wordclock/output`
    - Audio IO: Endpoint `/api/audio/levels` will be replaced by `/api/audio/metering` while the internal properties `danteIn` and `danteOut` will be replaced by `aoIpIn` and `aoIpOut` respectively.

    - Audio IO: The `source` property for audio inputs `/api/audio/inputs/{inputId}` will be replaced by `inputSource`
    - Audio IO: At the audio outputs `/api/audio/outputs/{outputId}` the following properties will be replaced by adding new and deprecating the old ones:
        - `commandModeAudioNetwork` will be replaced by `aoIpEnableIfCommandIsDisabled` and `aoIpEnableIfCommandIsEnabled`
        - `commandModeMadi1` will be replaced by `madi1EnableIfCommandIsDisabled` and `madi1EnableIfCommandIsEnabled`
        - `commandModeMadi2` will be replaced by `madi2EnableIfCommandIsDisabled` and `madi2EnableIfCommandIsEnabled`
    - Firmware Update: The `danteVersion` property will be replaced by `aoIpVersion` in the firmware update status endpoint `/api/firmware/update/state`
- Mobile Devices + RF: The `ledBrightness` property for paired devices at `/api/mts/paired/all/{mtUid}` and RF antenna configuration at `/api/rf/antennas` will be replaced. Please use the new `connectedStateColor` property for paired devices and `ledColors` for RF antenna configuration to set custom LED colors for the device.
- SSC: The endpoint `/api/ssc/heartbeat` is going to be removed with the next major version of the API. Use `/api/device/time` instead.

Preliminary Features

> **i** These functionalities are not fully implemented yet and the behavior may change in future versions of the API even without deprecation announcements.

- Mobile Devices: (Sleep Mode) `sleep` property for paired devices `/api/mts/paired/all/{mtUid}`
- Level Recording: All endpoints and especially the schema definition for the level recording data structure are still subject to change.

Known Issues

- RF: Although the property `bandwidthMode` at `/api/rf/channels/{rfChannelId}` can be set to the value `10000` (10 MHz) the system will never enable the RF transmission, since it is not certified yet for operation with 10 MHz bandwidth and all license entitlements will block the usage. Please choose a different bandwidth allowed with your license entitlement. Current RF restrictions can be read at `/api/rf/restrictions`.
- Mobile Devices + Audio IO: There is no dedicated audio mute per Mobile Device nor per Audio I/O available yet. We assume that the audio muting will be done outside the Spectera system e.g. on a mixing console.

OpenAPI specification 18.3

All HTTPs methods, parameters and responses for Spectera at a glance.



Firefox Enhanced Tracking Protection (ETP), especially in “Strict” mode, can block scripts or cookies required by Swagger UI, causing it not to load, showing “Fetch error” messages, or breaking the “Try it out” function. To resolve this, disable Enhanced Tracking Protection for the specific Swagger page only:

- Click the shield icon to the left of the URL in the Firefox address bar.
- Turn off the **Enhanced Tracking Protection** toggle for this site.
- Wait for the page to reload automatically and verify that Swagger UI works as expected.

The OpenAPI specification is only available online and can be accessed through the following link: Sennheiser 3rd party API

Level recording schema specification 1.1

The FBS schema view provides a structured representation of the FBS data model for analysis and integration.

You can download the FBS schema here.

```
namespace Lr;

// Schema Version: v1.1.0
table SchemaVersion {
  major:ubyte;
  minor:ubyte;
  patch:ubyte;
}

enum MdState:byte {Connected, Updating, Syncing, Sleeping, Disconnected}
enum HeadphonePlugState:byte {NotAvailable, Unplugged, Plugged}
enum AntennaPort:byte {A,B,C,D}
enum RfChannelId:byte {_1, _2}
enum InputSource:byte {Madi1, Madi2, AoIP}
enum OutputMode:byte {On, Off, Mute, Talk}
enum AntennaState:byte {Initialized,
                        Updating,
                        Connected,
                        Error,
                        Unconnected}
enum AntennaBinding:byte {RfChannel1, RfChannel2, Scan, NotBound}
enum RfBandwidth:byte { _6MHz, _8MHz, _10MHz}
enum RfState:byte {RfActive, RfMuted}
enum PsuState:byte {Connected, Disconnected, Unconnected}
enum SystemState:byte {Initializing, Normal, Warmup, Cooldown, FactoryReset, Standby,
 CriticalFailure}
enum TemperatureStateOverall:byte {Low, Normal, High, Critical}
enum FanError:byte {Ok, Error1, Error2, Broken}
enum AoipType:byte {None, Unknown, Dante, Ravenna} // Type of audio over IP (AoIP)
 interface: None=no AoIP interface, Unknown=unknown AoIP interface, Dante=Dante AoIP
 interface, Ravenna=Ravenna AoIP interface
enum MadiModuleType:byte {None, Unknown, Coax, OpticalMultimode} // Status of MADI slot
 module: None=no MADI module found, Unknown=unknown MADI module found, Coax=BNC Coax
 MADI module found, OpticalMultimode=Optical Multimode MADI module found
enum SampleRateClassHz:byte {_0, _16000, _22050, _24000, _32000, _44100, _48000, _64000,
 _88200, _96000, _128000, _176400, _192000} // Sample rate classes in Hz
enum SampleRateClassHzOutput:byte {_0, _48000, _96000} // Possible output sample rate
 classes in Hz
// Status of audio clock input signal
enum ClockStatus:byte {
  NoToggle,       // Input signal is not toggling or too slow
  TooFast,        // Input signal is too fast
  Mismatch,       // Input signal is detected but could not be used (only for MADI)
  Unlocked,       // Input signal is toggling but PLL is not locked (yet)
  Locked,         // Input signal is valid and PLL is locked
  LoopThrough     // Input signal is looped through to the output (only for Word Clock,
 only in non-operational state)
}
// Possible audio clock sources
enum OutputClockSource:byte {
  Internal48kHz,  // Leader mode, use internal 48 kHz
  Internal96kHz,  // Leader mode, use internal 96 kHz
  WordclockIn,    // Follow Word Clock BNC input
```

```
  AoIp,           // Follow Audio over IP
  Madi1In,        // Follow MADI 1 input
  Madi2In         // Follow MADI 2 input
}
enum CommandState:byte {Unknown, NotAvailable, Released, Pressed, Unlatched, Latched}
enum CommandBehavior:byte {Disabled, Momentary, Latching}
enum MicLineSelection:byte {Auto, Mic, Line}
enum MicLineSelectionAutoValue:byte {Unknown, Mic, Line}

// **** Base Station ****

struct PsuStates {
  state:[PsuState:2];
}
struct FanErrors {
  error:[FanError:3];
}
table BaseStation {
  psu_states:PsuStates;
  system_state:SystemState;
  temperature_state_overall:TemperatureStateOverall;
  fan_errors:FanErrors;
}

// **** Audio Metering ****

struct AudioMeteringPerChannel {
  peak:[ubyte:64]; // 64 times per frame, [0..255] -> [-127.5..0] in 0.5 dB steps
  rms:[ubyte:64];  // 64 times per frame, [0..255] -> [-127.5..0] in 0.5 dB steps
}

struct AudioMetering {
  madi1_in:[AudioMeteringPerChannel:32];
  madi2_in:[AudioMeteringPerChannel:32];
  aoip_in:[AudioMeteringPerChannel:32];
  madi1_out:[AudioMeteringPerChannel:32];
  madi2_out:[AudioMeteringPerChannel:32];
  aoip_out:[AudioMeteringPerChannel:32];
}

// **** Mobile Device ****

struct IemLqi {
  value:ubyte; // 3 bit, once per frame
}

struct MicLqi {
  value:[ubyte:64]; // 3 bit, 64 times per frame
}

struct MicRssiPerAntenna {
  value:[byte:64]; // [-100..20]dBm, 64 times per frame
}

struct MicRssi {
  antenna:[MicRssiPerAntenna:4]; // 4 antennas
}

struct ControlRssi {
  value:byte; // [-100..20]dBm, once per frame
```

```
}

struct DominantAntenna {
  value:AntennaPort;
}

struct InterferenceSeverityMd {
  value:ubyte; // 3 bit, 0: none, 1: low, 2: mid, 3: high, 7: not available
}

table Battery {
  percentage:ubyte; // 0..100
  runtime:ushort; // in minutes
  is_low:ubyte;
}

table IemLink {
  audiolink_mode:uint32;
  input_channel:uint8;
  input_source:InputSource;
  lqi:IemLqi;
}

table MicChannel {
  output_channel:uint8;
  output_mode_madi1:OutputMode;
  output_mode_madi2:OutputMode;
  output_mode_aoip:OutputMode;
}

table MicLinkSek {
  audiolink_mode:uint32;
  channels:[MicChannel];
  lqi:MicLqi;
  command_state:CommandState;
  mic_line_selection_auto_value:MicLineSelectionAutoValue;
}

table MicLinkSkm {
  audiolink_mode:uint32;
  channels:[MicChannel];
  lqi:MicLqi;
  command_state:CommandState;
}

table Sek {

  // fields that are always available, independent from the state.
  uid:uint32;
  rf_channel_id:RfChannelId;
  name:string;
  state:MdState;
  headphone_volume:ubyte; // [0..255] -> [-100..27.5] in 0.5 dB steps
  headphone_plug_state:HeadphonePlugState;

  // fields that are present if MdState is not Disconnected or Sleeping
  battery:Battery;
  control_rssi:ControlRssi;
  dominant_antenna:DominantAntenna;
  interference_severity:InterferenceSeverityMd;
```

```
  mic_rssi:MicRssi;

  iem_link:IemLink; // present if a iem link is configured
  mic_link:MicLinkSek; // present if a mic link is configured

  command_behavior:CommandBehavior;
  mic_line_selection:MicLineSelection;
}

table Skm {

  // fields that are always available, independent from the state.
  uid:uint32;
  rf_channel_id:RfChannelId;
  name:string;
  state:MdState;

  // fields that are present if MdState is not Disconnected or Sleeping
  battery:Battery;
  control_rssi:ControlRssi;
  dominant_antenna:DominantAntenna;
  interference_severity:InterferenceSeverityMd;
  mic_rssi:MicRssi;

  mic_link:MicLinkSkm; // present if a mic link is configured

  command_behavior:CommandBehavior;
}

// **** Antenna ****

struct AntennaTemperature {
  temperature:byte; // [-55 .. 150] in degrees Celsius
  warning_high_temperature:bool;
}

struct ScanSlot {
  input_power:[byte:64]; // [-100..20]dBm, 64 times per frame
  residual_power:[byte:64]; // [-100..20]dBm, 64 times per frame
  total_severity:[ubyte:64]; // 2bit, 64 times per frame
  residual_severity:[ubyte:64]; // 2bit, 64 times per frame
}

table Antenna {

  // fields that are always available, independent from the state
  port:AntennaPort;
  state:AntennaState;
  binding:AntennaBinding;
  warning_packet_error:bool;

  temperature:AntennaTemperature; // present if AntennaState is not Unconnected or Error
  scanslot:ScanSlot; // present if AntennaState is Initialized and bound to an RF
 channel
}

// **** RF Channel ****

table RfChannel {
```

```
  id:RfChannelId;
  center_frequency:uint32;
  bandwidth:RfBandwidth;
  state:RfState;
}

// **** Audio Interfaces ****

table Aoip {
  type:AoipType;
  clock_status:ClockStatus;
  sample_rate_class:SampleRateClassHz;
  mute:bool;
}

table WclkInput {
  clock_status:ClockStatus;
  sample_rate_class:SampleRateClassHz;
}

table WclkOutput {
  clock_source:OutputClockSource;
  clock_source_status:ClockStatus;
  fallback_clock_active:bool; // True if selected input source is not valid and fallback
 to internal rate was necessary
  sample_rate_class:SampleRateClassHzOutput;
}

table WCLK {
  input:WclkInput;
  output:WclkOutput;
}

table MadiInput {
  clock_status:ClockStatus;
  sample_rate_class:SampleRateClassHz;
  mute:bool;
  channels:uint8; // Detected number of MADI input channels (0 if ModuleType is None or
 Status is not Locked)
  fewer_channels_than_used:bool; // True if at least one audio link is configured to a
 MADI input channel that is not available
}

table MadiOutput {
  clock_source:OutputClockSource;
  clock_source_status:ClockStatus;
  fallback_clock_active:bool; // True if selected input source is not valid and fallback
 to internal rate was necessary
  sample_rate_class:SampleRateClassHzOutput;
  mute:bool;
  channels:uint8; // Number of MADI output channels (0 if ModuleType is None)
}

table MADI {
  module_type:MadiModuleType;
  input:MadiInput;
  output:MadiOutput;
}

table AudioInterfaces {
  aoip:Aoip;
```

```
  wclk:WCLK;
  madi1:MADI;
  madi2:MADI;
}

// **** Root Table ****

table LRFrame {
  index:uint64;
  time_stamp:uint64; // unix time in ms

  audio_metering:AudioMetering;
  seks:[Sek];
  skms:[Skm];
  antennas:[Antenna];
  rf_channels:[RfChannel];
  schema_version:SchemaVersion;
  base_station:BaseStation;
  audio_interfaces:AudioInterfaces;
}

root_type LRFrame;
```

Spectera OpenAPI version 18.1

Release Notes Version 18.1

Latest information on the Spectera API, including detailed descriptions of new features, breaking changes, upcoming deprecations and known issues.

Version 18.1.0

New Features

- Command Mode: New API endpoints and properties for controlling the routing and behavior of Mobile Devices and Audio I/O for command mode. See the OpenAPI specification and description for details.
- Level Recording (Preliminary): New API endpoints for recording and retrieving audio levels of Mobile Devices and Audio I/O, including a new schema definition for the level recording data structure. See the OpenAPI specification and description for details.

Improvements

- Mobile Devices: New property `swUpdateNotPossibleReason` for paired devices to provide more information on why a software update is not possible for a specific device. See the OpenAPI specification and description for details.
- Mobile Devices + RF (Custom LED Colors): With the new `connectedStateColor` property for paired devices `/api/mts/paired/all/{mtUid}` and `ledColors` for RF antenna configuration `/api/rf/antennas` it is now possible to set custom LED colors for the device. This will replace the previous `ledBrightness` property.
- Device: It's now possible to retrieve the current system time with the endpoint `/api/device/time` which can be used for better synchronization of client applications with the Spectera system e.g. for Level Recording timestamps.

Breaking Changes

- Mobile Devices: The deprecated `connected` property for paired devices has been removed now - Use the `state` property with values "Connected" and "Disconnected" instead
- Mobile Devices (Command Mode): The old preliminary Command Mode feature has been changed slightly which leads to some breaking changes for the properties `commandBehavior` and `commandState` for paired devices at `/api/mts/paired/all/{mtUid}` .
- Audio IO (Command Mode): The properties `commandModeAudioNetwork` , `commandModeMadi1` and `commandModeMadi2` for Audio Outputs `/api/audio/outputs/{outputId}` can not be used for controlling the command mode behavior anymore.
    - Although the command mode was already introduced in version 17.0 as preliminary feature, the properties mentioned above will be deprecated and behave differently already now. The existing values are mapped to

the new `*EnableIfCommandIsDisabled` properties only. The values `Mute` and `Talk` are converted to `On` to prevent unintentional audio muting. To activate the `Mute` and `Talk` command behavior the new `*EnableIfCommandIsEnabled` properties must be used.

Upcoming Deprecations

> **i** These functionalities are still available in this version but will be removed in the next major version. Please update your implementation to use the new alternatives as soon as possible to avoid disruptions when upgrading to the next major version.

- Audio Interfaces + Audio IO: As preparation for upcoming new audio interface the name of the interface `Dante` and `AudioNetwork` will be replaced with the more generic name `AoIP` (Audio over IP). The following endpoints and properties will be replaced by adding new and removing the old ones. The old endpoints and properties are marked as deprecated until they are removed with the next major version:
  - Audio Interfaces: Endpoints `/api/audio/interface/audionetwork/status` and `/api/audio/interface/audionetwork/clock/input/status` will be replaced by `/api/audio/interface/aoip/status` and `/api/audio/interface/aoip/clock/input/status`
  - Audio Interfaces: The `outputStatus/clockSource` property for the following endpoints will be replaced by `outputStatus/outputClockSource`
    - `/api/audio/interface/madi1/status`
    - `/api/audio/interface/madi2/status`
    - `/api/audio/interface/wordclock/status`
  - Audio Interfaces: The `clockSource` property for the following endpoints will be replaced by `outputClockSource`
    - `/api/audio/interface/madi1/output`
    - `/api/audio/interface/madi2/output`
    - `/api/audio/interface/wordclock/output`
  - Audio IO: Endpoint `/api/audio/levels` will be replaced by `/api/audio/metering` while the internal properties `danteIn` and `danteOut` will be replaced by `aoIpIn` and `aoIpOut` respectively.
  - Audio IO: The `source` property for audio inputs `/api/audio/inputs/{inputId}` will be replaced by `inputSource`

    - Audio IO: At the audio outputs `/api/audio/outputs/{outputId}` the following properties will be replaced by adding new and deprecating the old ones:
        - `commandModeAudioNetwork` will be replaced by `aoIpEnableIfCommandIsDisabled` and `aoIpEnableIfCommandIsEnabled`
        - `commandModeMadi1` will be replaced by `madi1EnableIfCommandIsDisabled` and `madi1EnableIfCommandIsEnabled`
        - `commandModeMadi2` will be replaced by `madi2EnableIfCommandIsDisabled` and `madi2EnableIfCommandIsEnabled`
    - Firmware Update: The `danteVersion` property will be replaced by `aoIpVersion` in the firmware update status endpoint `/api/firmware/update/state`
- Mobile Devices + RF: The `ledBrightness` property for paired devices at `/api/mts/paired/all/{mtUid}` and RF antenna configuration at `/api/rf/antennas` will be replaced. Please use the new `connectedStateColor` property for paired devices and `ledColors` for RF antenna configuration to set custom LED colors for the device.
- SSC: The endpoint `/api/ssc/heartbeat` is going to be removed with the next major version of the API. Use `/api/device/time` instead.

Preliminary Features

> **i** These functionalities are not fully implemented yet and the behavior may change in future versions of the API even without deprecation announcements.

- Mobile Devices: (Sleep Mode) `sleep` property for paired devices `/api/mts/paired/all/{mtUid}`
- Level Recording: All endpoints and especially the schema definition for the level recording data structure are still subject to change.

Known Issues

- RF: Although the property `bandwidthMode` at `/api/rf/channels/{rfChannelId}` can be set to the value `10000` (10 MHz) the system will never enable the RF transmission, since it is not certified yet for operation with 10 MHz bandwidth and all license entitlements will block the usage. Please choose a different bandwidth allowed with your license entitlement. Current RF restrictions can be read at `/api/rf/restrictions`.
- Mobile Devices + Audio IO: There is no dedicated audio mute per Mobile Device nor per Audio I/O available yet. We assume that the audio muting will be done outside the Spectera system e.g. on a mixing console.

OpenAPI specification 18.1

All HTTPs methods, parameters and responses for Spectera at a glance.

Firefox Enhanced Tracking Protection (ETP), especially in “Strict” mode, can block scripts or cookies required by Swagger UI, causing it not to load, showing “Fetch error” messages, or breaking the “Try it out” function. To resolve this, disable Enhanced Tracking Protection for the specific Swagger page only:

- Click the shield icon to the left of the URL in the Firefox address bar.
- Turn off the **Enhanced Tracking Protection** toggle for this site.
- Wait for the page to reload automatically and verify that Swagger UI works as expected.

The OpenAPI specification is only available online and can be accessed through the following link: Sennheiser 3rd party API

Level recording schema specification 1.0

The FBS schema view provides a structured representation of the FBS data model for analysis and integration.

You can download the FBS schema here.

```
namespace Lr;

enum MdState:byte {Connected, Updating, Syncing, Sleeping, Disconnected}
enum HeadphonePlugState:byte {NotAvailable, Unplugged, Plugged}
enum AntennaPort:byte {A,B,C,D}
enum RfChannelId:byte {_1, _2}
enum InputSource:byte {Madi1, Madi2, AoIP}
enum OutputMode:byte {On, Off, Mute, Talk}
enum AntennaState:byte {Initialized,
Updating,
Connected,
Error,
Unconnected}
enum AntennaBinding:byte {RfChannel1, RfChannel2, Scan, NotBound}
enum RfBandwidth:byte { _6MHz, _8MHz, _10MHz}
enum RfState:byte {RfActive, RfMuted}

// **** Audio Metering ****

struct AudioMeteringPerChannel {
peak:[ubyte:64]; // 64 times per frame, [0..255] -> [-127.5..0] in 0.5 dB
steps
rms:[ubyte:64];  // 64 times per frame, [0..255] -> [-127.5..0] in 0.5 dB
steps
}

struct AudioMetering {
madi1_in:[AudioMeteringPerChannel:32];
madi2_in:[AudioMeteringPerChannel:32];
aoip_in:[AudioMeteringPerChannel:32];
madi1_out:[AudioMeteringPerChannel:32];
madi2_out:[AudioMeteringPerChannel:32];
aoip_out:[AudioMeteringPerChannel:32];
}

// **** Mobile Device ****

struct IemLqi {
value:ubyte; // 3 bit, once per frame
}

struct MicLqi {
value:[ubyte:64]; // 3 bit, 64 times per frame
}

struct MicRssiPerAntenna {
value:[byte:64]; // [-100..20]dBm, 64 times per frame
}

struct MicRssi {
antenna:[MicRssiPerAntenna:4]; // 4 antennas
```

```
}

struct ControlRssi {
value:byte; // [-100..20]dBm, once per frame
}

struct DominantAntenna {
value:AntennaPort;
}

struct InterferenceSeverityMd {
value:ubyte; // 2 bit
}

table Battery {
percentage:ubyte; // 0..100
runtime:ushort; // in minutes
is_low:ubyte;
}

table IemLink {
audiolink_mode:uint32;
input_channel:uint8;
input_source:InputSource;
lqi:IemLqi;
}

table MicChannel {
output_channel:uint8;
output_mode_madi1:OutputMode;
output_mode_madi2:OutputMode;
output_mode_aoip:OutputMode;
}

table MicLink {
audiolink_mode:uint32;
channels:[MicChannel];
lqi:MicLqi;
}

table Sek {

// fields that are always available, independent from the state.
uid:uint32;
rf_channel_id:RfChannelId;
name:string;
state:MdState;
headphone_volume:ubyte;
headphone_plug_state:HeadphonePlugState;

// fields that are present if MdState is not Disconnected or Sleeping
battery:Battery;
control_rssi:ControlRssi;
dominant_antenna:DominantAntenna;
interference_severity:InterferenceSeverityMd;
mic_rssi:MicRssi;

iem_link:IemLink; // present if a iem link is configured
mic_link:MicLink; // present if a mic link is configured
}
```

```
            table Skm {

            // fields that are always available, independent from the state.
            uid:uint32;
            rf_channel_id:RfChannelId;
            name:string;
            state:MdState;

            // fields that are present if MdState is not Disconnected or Sleeping
            battery:Battery;
            control_rssi:ControlRssi;
            dominant_antenna:DominantAntenna;
            interference_severity:InterferenceSeverityMd;
            mic_rssi:MicRssi;

            mic_link:MicLink; // present if a mic link is configured
            }

            // **** Antenna ****

            struct AntennaTemperature {
            temperature:byte; // [-55 .. 150] in degrees Celsius
            warning_high_temperature:bool;
            }

            struct ScanSlot {
            input_power:[byte:64]; // [-100..20]dBm, 64 times per frame
            residual_power:[byte:64]; // [-100..20]dBm, 64 times per frame
            total_severity:[ubyte:64]; // 2bit, 64 times per frame
            residual_severity:[ubyte:64]; // 2bit, 64 times per frame
            }

            table Antenna {

            // fields that are always available, independent from the state
            port:AntennaPort;
            state:AntennaState;
            binding:AntennaBinding;
            warning_packet_error:bool;

            temperature:AntennaTemperature; // present if AntennaState is not
Unconnected or Error
            scanslot:ScanSlot; // present if AntennaState is Initialized and bound to an
RF channel
            }

            // **** RF Channel ****

            table RfChannel {
            id:RfChannelId;
            center_frequency:uint32;
            bandwidth:RfBandwidth;
            state:RfState;
            }

            // **** Root Table ****

            table LRFrame {
```

```
            index:uint64;
            time_stamp:uint64; // unix time in ms

            audio_metering:AudioMetering;
            seks:[Sek];
            skms:[Skm];
            antennas:[Antenna];
            rf_channels:[RfChannel];
            }

            root_type LRFrame;
```

Spectera OpenAPI version 17.0

Release Notes Version 17.0

Latest information on the Spectera API, including detailed descriptions of new features, breaking changes, upcoming deprecations and known issues.

Version 17.0.1

Improvements

- OpenAPI Specification (YAML) completely reviewed and refactored on description and documentation for public release

Version 17.0.0

New Features

- Since this is the first official public API version, we don't have any new features to announce

Breaking Changes

- Since this is the first official public API version, we don't have any breaking changes to announce

Upcoming Deprecations

- Mobile Devices: The boolean `connected` property for paired devices is going to be removed with the next major version of the API - Use the `state` property with values "Connected" and "Disconnected" instead

Preliminary Features

- These functionalities are not fully implemented yet and the behavior may change in future versions of the API.
- Mobile Devices: (Sleep Mode) `sleep` property for paired devices `/api/mts/paired/all/{mtUid}`
- Mobile Devices: (Command Mode) `commandState` and `commandBehavior` properties for paired devices `/api/mts/paired/all/{mtUid}`
- Audio IO: (Command Mode) `commandModeAudioNetwork`, `commandModeMadi1` and `commandModeMadi2` properties for Audio Outputs `/api/audio/outputs/{outputId}:`

Known Issues

- RF: Although the property `bandwidthMode` at `/api/rf/channels/{rfChannelId}` can be set to the value `10000` (10 MHz) the system will never enable the RF transmission, since it is not certified yet for operation with 10 MHz bandwidth and all license entitlements will block the usage. Please choose a different bandwidth allowed with your license entitlement. Current RF restrictions can be read at `/api/rf/restrictions`.
- Mobile Devices + Audio IO: There is no dedicated audio mute per Mobile Device nor per Audio I/O available yet. We assume that the audio muting will be done outside the Spectera system e.g. on a mixing console.

OpenAPI Specification 17.0

All HTTPs methods, parameters and responses for Spectera at a glance.



Firefox Enhanced Tracking Protection (ETP), especially in “Strict” mode, can block scripts or cookies required by Swagger UI, causing it not to load, showing “Fetch error” messages, or breaking the “Try it out” function. To resolve this, disable Enhanced Tracking Protection for the specific Swagger page only:

- Click the shield icon to the left of the URL in the Firefox address bar.
- Turn off the **Enhanced Tracking Protection** toggle for this site.
- Wait for the page to reload automatically and verify that Swagger UI works as expected.

The OpenAPI specification is only available online and can be accessed through the following link: Sennheiser 3rd party API

MobileConnect



Sennheiser electronic SE & Co. KG | Am Labor 1 | 30900 Wedemark | Germany