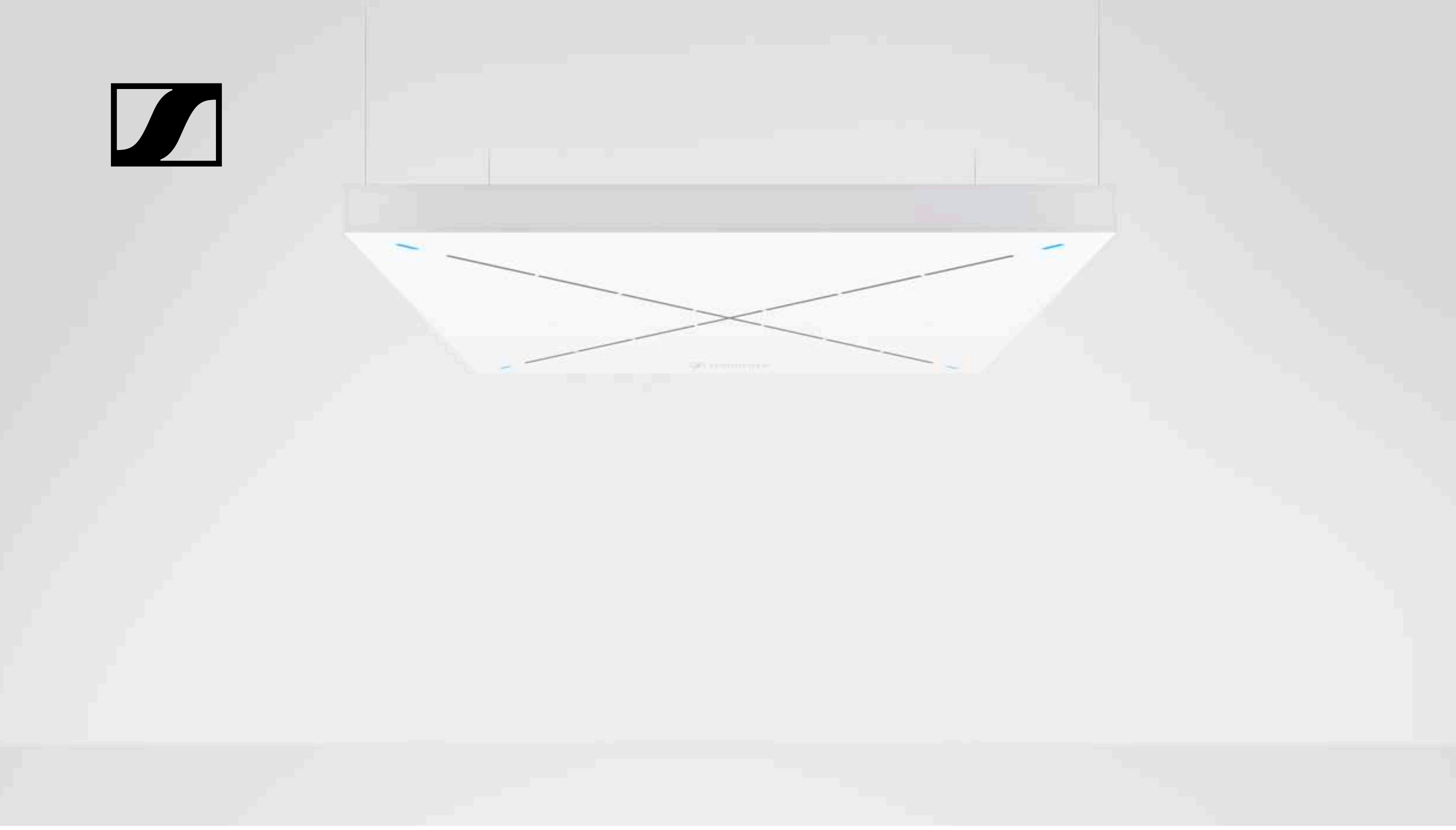

Sennheiser Sound Control Protocol (SSC)

Media control protocol

TeamConnect Ceiling 2

SENNHEISER

Sennheiser electronic GmbH & Co. KG

Am Labor 1, 30900 Wedemark, Germany, www.sennheiser.com
TI 1245 v1.1

Table of Contents

1. Introduction

Modern professional audio devices are designed as building blocks for large, complex systems.

Whereas audio signal paths have converged to industry standards a long time ago, driven by practical necessities, and only recently challenged by new transport technologies like Ethernet, the professional audio markets have not evolved a similar technological convergence in the area of remote, centralised control of systems of audio equipment (the notable historical exception being MIDI, which but has a limited scope and extensibility).

In this heterogeneous environment of diverging standards proposed by individual vendors as well as open communities, there is no existing self-evident solution to be found for the needs raised by designing professional Sennheiser audio equipment.

As a consequence, communication protocols implemented in Sennheiser products have so far been designed on a single-product or product-family basis. This has worked sufficiently well, up to the point that separately developed protocols start to concur in nexus devices or applications, like:

- Wireless Systems Manager (PC-based control application for wireless transmission)
- remote channel for future Sennheiser PRO microphones
- Media Control Systems (third party products, e.g., Crestron)
- A/V studio integration (third party products, e.g., Lawo)
- smartphone or tablet apps
- future centralised Sennheiser services

It has become evident that product-specific protocols fail to scale well in nexus products because of the added complexity in re-implementing the same remote control functionality from a customer point of view in a multitude of different backwards-compatible ways. It is not feasible to add more ever different technical solutions to the existing variety - the aim must be to define a reasonably future-proof protocol suitable for existing as well as envisioned products, devices, and services.

A broad market evaluation of existing technical solutions was performed in a joint Sennheiser PRO/IS working group. As a result, it turns out that Open Sound Control comes closest to the specific needs for an extensible, future-proof command, control, metering, and configuration protocol for Sennheiser products.

This document describes the specific adaption of Open Sound Control to Sennheiser use, “Sennheiser Sound Control”, SSC. The main other ingredient is JavaScript Object Notation (JSON), which enhances ease-of-use and the implementation complexity for small to smallest devices.

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

2. Open Sound Control Overview

Open Sound Control (OSC) is a protocol developed at The Center For New Music and Audio Technology (CNMAT) at University of California, Berkeley.

The OSC specification Version 1.1 is available from the Open Sound Control website at

http://www.opensoundcontrol.org/.

It is a very simple and very extensible protocol that can be implemented easily in embedded systems. It can be transported over IPv4 and IPv6 protocols using UDP packets and TCP streams.

Even very small PIC microcontrollers can handle OSC messages via projects such as MicroOSC from http://cnmat.berkeley.edu/research/uosc.

The OSC Schema defined by MicroOSC at:

http://cnmat.berkeley.edu/library/uosc_project_documentation/osc_address_schema was used as a starting point for some parts of the schema defined in this document.

OSC handles more advanced packet formats such as bundles of messages to be atomically executed at the same time with timestamps, as well as addresses with wildcards and array values.

2.1 JavaScript Object Notation Overview

JavaScript Object Notation (JSON) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. 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 the C-family of 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 4627 (MIME-type *application/json*).

JavaScript Object Notation (JSON) is a text format for the serialization of structured data. It is derived from the object literals of JavaScript, as defined in the ECMAScript Programming Language Standard, Third Edition.

JSON can represent four primitive types (strings, numbers, booleans, and null) and two structured types (objects and arrays).

A string is a sequence of zero or more Unicode characters.

An object is an unordered collection of zero or more name/value pairs, where a name is a string and a value is a string, number, boolean, null, object, or array.

An array is an ordered sequence of zero or more values.

The terms "object" and "array" come from the conventions of JavaScript.

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

3. 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".

3.1 Terminology

SSC Message	protocol unit of transmission
SSC Server	device or application that receives SSC messages, and replies to them
SSC Client	device, application or person that sends SSC messages
SSC Container	named entity containing SSC Methods or other Containers
SSC Method	named attribute or action callable on a SSC Server
SSC Address	full name of a SSC Method, including names of all enclosing Containers may be represented by a JSON object hierarchy
SSC Address Tree	a JSON object hierarchy consisting of one or more SSC Addresses
SSC Address Space	hierarchical tree comprising all the SSC Addresses of a SSC Server
SSC Method Call	SSC Message requesting execution of a SSC Method
SSC Method Arguments	arguments included in a SSC Method Call
SSC Method Reply	SSC Message send by SSC Server as result of a Method Call
binary OSC	the binary OSC encoding as opposed to JSON-based SSC
restricted SSC Server	a SSC Server that doesn‘t implement some optional parts of this specification

4. SSC Data Structure Specification

4.1 Applying JSON to the OSC device model

OSC models the controlled device as a tree-shaped hierarchy of *methods*, with the method *addresses* constructed from the names of all the nodes in the hierarchy, written like a file path.

`/`	container at address "/"
`    out1/`	container at address "/out1/"
`        xlr1/`	container at address "/out1/xlr1/"
`            gain 5`	address "/out1/xlr1/gain": method with numeric argument
`            mute true`	address "/out1/xlr1/mute": method with a boolean argument
`            ...`	more methods of "/out1/xlr1"
`        xlr2/`	container at address "/out1/xlr2/"
`        ...`	methods of “/out1/xlr2”
`    out2/`	container at address "/out2/"
`        ...`	methods of "/out2"
`    ...`	more methods and containers of "/"

JSON allows to model that structure as a hierarchy of *JSON objects*.

`{`	root object
`    "out1": {`	object "out1"
`        "xlr1": {`	object "out1.xlr1"
`            "gain": 5,`	numerical property "out1.xlr1.gain"
`            "mute": true,`	boolean property "out1.xlr1.mute"
`            ...`	more properties of "out1.xlr1"
`        },`	
`        "xlr2": {`	object "out1.xlr2"
`            ...`	properties of "out1.xlr2"
`        },`	
`    "out2": {`	object "out2"
`        ...`	properties of "out2"
`    },`	
`    ...`	more properties and objects of the root object
`}`	

The OSC Method Address (like „/out1/xlr2/gain”) is interpreted as a property path navigating through the hierarchy of JSON objects. The value of each property MUST be either a primitive JSON data type, or a JSON array. Rationale: This allows to clearly separate SSC Method Addresses from SSC Method Arguments at JSON parser level without knowledge of the underlying method address tree.

The resulting JSON tree structure of hierarchical objects, the SSC *Address Space*, is tailored to describe the functionality of a specific SSC Server, in the same way as foreseen by OSC.

In JSON it is possible to serialise the complete state of all properties in the tree to a closed form, thus describing the complete state of the SSC Server. In this way, JSON can be used as an excellent extensible data format for configuration files, or for scripting applications, which drive a system of SSC Servers through a sequence of programmed configurations.

For command and control applications it is desirable to access single properties independently. This can be achieved in JSON syntax by the simple convention, that all the properties of an SSC Server that are not mentioned in a JSON message are left unchanged.

In this way, applied to the example above, the JSON form

```
    { "out1": { "xlr1": { "gain": 5 } } }
```

can be understood as an SSC Method Call of the SSC Method „/out1/xlr1/gain“ with the argument 5, presumably to set the gain to that level, or as an SSC Method Reply message stating the current gain level.

4.2 JSON Message Transaction Syntax

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

- Prefix „TX:“ indicates an SSC Message that an SSC Client is sending to an SSC Server.
- Prefix „RX:“ indicates an SSC Message that the SSC Server will send back to the Client.
- An SSC-Message is written verbatim, enclosed by curly brackets { }.

A transaction to set the gain of “xlr2” of “out1” to -10 then looks like this:

```
    TX: { "out1": { "xlr2": { "gain": -10 }}}
    RX: { "out1": { "xlr2": { "gain": -10 }}}
```

Note that the execution of the method results in a method reply message, which for simple property setters states the actual value of the property resulting from executing the message.

The resulting value may be different from the supplied argument, e.g., for a read-only property, or if the argument is out of range, and the device may adapt it to the allowed range (this is not considered as an error):

```
    TX: { "out1": { "xlr2": { "gain": -10000 }}}
    RX: { "out1": { "xlr2": { "gain": -15 }}}
```

Getter-methods, which request the value of a property from the SSC Server, are realised by supplying the special JSON value null as argument to method sent to the address of the property:

```
    TX: { "out1": { "xlr2": { "gain": null }}}
    RX: { "out1": { "xlr2": { "gain": -15 }}}
```

Compared to binary OSC, the JSON syntax is slightly more verbose for single attribute settings, but this is compensated when multiple attributes are set in the same transaction:

```
    TX: { "out1": { "xlr2": { "gain": -10, "mute": false }}}
    RX: { "out1": { "xlr2": { "gain": -10, "mute": false }}}
```

SSC Address Patterns are an optional feature for an SSC Server. They allow transactions like the following, to presumably to mute all outputs at once:

```
    TX: { “out1”: { “*”: { “mute”: true }}}
    RX: { “out1”: { “xlr1”: { “mute”: true },
                    “xlr2”: { “mute”: true }}}
```

To facilitate true interactive use, an extra-placable SSC Server is introduced as an implementation option.

4.3 SSC JSON Message Syntax

4.3.1 Elementary data types

All SSC data is composed of the primitive JSON data types:

- string: a sequence of zero or more Unicode characters in UTF-8 encoding, wrapped in double quotes, using backslash escapes. A character is represented as a single character string. Binary zero bytes can be included in a string using Unicode escape notation: „\u0000“.
- number: a number in conventional „scientific“ notation. `0, 42, -23, 3.141259, 1.0e+100` are all valid numbers. A Restricted SSC Server MAY reject non-integer numeric arguments, or it MAY adapt them by silently converting them to integer values.
- `true`: the boolean true value.
- `false`: the boolean false value.
- `null`: indicates a missing value; used as pseudo argument for getter-methods.

The SSC Server MAY auto-convert elementary data types without further indication to their specific purpose. The client is usually informed about the actual value used by the SSC Server in the response to the SSC method execution.

If the server auto-converts the data type, it MUST follow these conversion rules:

- string to number: String is parsed for leading whitespace, which is skipped, then for a numerical part. Any remaining non-numerical tailing characters are ignored. Completely non-numerical strings convert to zero. The exact behaviour MUST have the same result as calling the C standard function `strtod()`.
- string to boolean: Any non-empty string is true, an empty string is `false`.
- number to string: a string representation of the number suitable for interpretation by `strtod()` is used.
- number to boolean: Number zero is `false`, everything else is `true`.
- boolean to string: true results in “`true`”, false in an empty string “”.
- boolean to number: true is `1`, false is `0`.

4.3.2 SSC Messages

A Message is the protocol unit of transmission. Any application that sends SSC Messages is an SSC Client, any application that receives SSC Messages is an SSC Server.

An SSC Message MUST be sent as a single closed JSON form describing a JSON object. Extra whitespace between the elements of the message MUST be ignored by the receiver.

This means that every SSC Message is enclosed in a pair of curly brackets `{ }`.

The length of an SSC Message is variable. If the underlying transport protocol is packet-based, like UDP/IP or ZigBee, then exactly one SSC Message SHOULD be contained in one transport packet. If the underlying transport protocol is a byte-stream, like TCP/IP or a serial link, then SSC Messages MUST be terminated, additionally to the grouping provided by the JSON syntax, with Message Separator Characters specific to the transport. The message separator characters MUST NOT be able to occur unescaped in the non-ignored contents of the packet. Compare section [TCP/IP].

4.3.3 SSC Addresses

Every SSC Server implements a set of SSC Methods. SSC Methods are the potential destinations of SSC Messages received by the SSC Server, and correspond to each of the points of control that the application makes available. “Invoking” an SSC Method is analogous to a procedure call; it means supplying the method with arguments and causing the method’s effect to take place. The SSC Server MUST respond to each received SSC Message by sending an SSC Method Reply Message to the originating SSC Client.

An SSC Server’s SSC Methods are arranged in a tree structure called an SSC Address Space. The leaves of this tree are the SSC Methods and the branch nodes are called SSC Containers. An SSC Server’s SSC Address Space MAY be dynamic; that is, its contents and shape MAY change over time.

Each SSC Method and each SSC Container other than the root of the tree MUST have a symbolic name which MUST be composed entirely of printable ASCII characters other than the following:

" "	space,	ASCII 32
"	double quote,	ASCII 34
#	number sign,	ASCII 35
*	asterisk,	ASCII 42
,	comma,	ASCII 44
/	slash,	ASCII 47
:	colon,	ASCII 58
?	question mark,	ASCII 63
[	open bracket,	ASCII 91
]	close bracket,	ASCII 93
{	open curly brace,	ASCII 123
}	close curly brace,	ASCII 125

The SSC Address of an SSC Method is a symbolic name giving the full path to the SSC Method in the SSC Address Space, starting from the root of the tree. An SSC Method’s SSC Address begins with the character „/” (forward slash), followed by the names of all the containers, in order, along the path from the root of the tree to the SSC Method, separated by forward slash characters, followed by the name of the SSC Method. The syntax of SSC Addresses was chosen to match the syntax of URLs. The SSC address syntax SHOULD be used in documentation, but it SHOULD NOT be used as an argument to other SSC Methods; the JSON syntax of hierarchical objects SHOULD be used instead.

SSC Methods MAY be overloaded with respect to their arguments: the SSC Server may execute the method in different ways depending on the arguments given.

SSC Methods MAY also be overloaded with respect to their Address: the SSC Server may execute a different SSC Method instead, and reply with an SSC Method Reply to that different SSC Method Address („aliased” SSC Methods). Example: a wireless receiver might report the battery charge level of the wireless transmitter either as a lifetime or as a percentage, and it might respond to a general “battery state” SSC Method Address either by executing the lifetime or the percentage Method, depending on the circumstances.

An SSC Method may be invoked with an empty argument list by supplying the JSON null value. This kind of SSC Method call SHOULD normally have the semantics of a query resulting in the current value of the property addressed by the method, without further side effects. SSC Methods that change the state of an SSC Server SHOULD normally have arguments.

Example:

- query current gain of XLR2 output of OUT1 module:
```
TX: { “out1”: { “xlr2”: { “gain”: null }}}
RX: { “out1”: { “xlr2”: { “gain”: -10 }}}
```
- change gain of XLR2 output of OUT1 module (note that the server adapts the value):
```
TX: { “out1”: { “xlr2”: { “gain”: -10000 }}}
RX: { “out1”: { “xlr2”: { “gain”: -15 }}}
```

5. SSC subscriptions - /osc/state/subscribe

A subscription request is sent by a client to a server for an address pattern to subscribe to. The SSC Server normally accepts the subscription request, and remembers that the requesting client wishes to be notified about value changes of the subscribed addresses.

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

The SSC Server MUST send an initial subscription notification to the client, which contains the result of calling the subscribed SSC Methods immediately with null-argument when the subscription request is handled. This initial notification MAY be bundled with the reply to the subscription request itself.

Each subscription notification MUST have identical contents to the reply to an imagined SSC Method invocation with null-argument to the subscribed SSC Method Address at the time that the notification is sent.

The SSC Client MAY bundle a call to `/osc/xid` with the subscription request. If an xid is supplied, a reply to `/osc/xid` MAY be bundled with each subscription notification, with the xid of the reply identical to that supplied by the client.

The SSC Server MUST send value changes of the subscribed addresses to the SSC Client. By default, the SSC Server will send subscription notifications if and only if the subscribed addresses change in value. The SSC Client can modify this behaviour by supplying optional parameters with the subscription request, allowing to either throttle the rate of notifications, or stimulate additional periodic notifications even if the subscribed addresses do not change in value.

Every subscription is specific to the connection between SSC Client and SSC Server. Also each SSC Method can only be subscribed once per connection. This means, that if an SSC Client requests a subscription which is already subscribed by that client on that connection, then the SSC Server MUST treat this as if the existing subscription was silently terminated and immediately requested anew.

5.1 Subscription notification rate parameters

Optional subscription request parameters related to notification rate:

- “min” - minimum notification period (ms), 0=none, default 0
- “max” - maximum notification period (ms), 0=none, default 0
- “bw” - maximum bandwidth for replies (byte/s), 0=unlimited, default 0

If “min” is 0, then notifications are not sent when a subscribed address changes in value, they are only sent based on the “max” period. If “min” is greater than 0, notifications are sent after the specified time duration has elapsed, even if the value of the subscribed address is unchanged.

If “max” is 0, then notifications are only sent when a value changes, or based on the “min” period. If “max” is greater than 0, then notifications are sent not earlier before the specified time duration has elapsed, even if the subscribed address changes value in the meantime.

5.2 Subscription cancelling and expiration

The SSC Server MUST terminate a subscription in these cases:

- the subscribed client cancels the subscription explicitly
- a maximum number of notifications has been sent
- a maximum lifetime relating to the begin of the subscription expires
- the SSC Client closes the connection
- the transport layer of the SSC connection signals a communication error

If the SSC Server decides to terminate the connection because the lifetime or notification count expires, then it MUST inform the SSC Client by sending an error reply “310 – subscription terminated” to the SSC address that terminates subscription together with or immediately after the last subscription notification.

Optional subscription request parameters related to termination:

- “cancel” „true“ cancels the subscription (default false).
- “count” maximum number of notifications to send, default 1000
- “lifetime” maximum lifetime (s) of the subscription, default 10s

The SSC Client may renew a subscription at any time, thereby resetting all of the lifetime limitations. To renew a subscription, the SSC Client re-requests it; there’s no difference between an initial subscription request and a renewal request.

5.3 Subscribing to multiple addresses

The SSC Client MAY request multiple subscriptions in a single request; either by providing them explicitly as SSC Address Tree, or by specifying address patterns as subscription addresses, or even both in the same request.

The SSC Server MAY either treat all those subscription requests separately, as if the addresses had all been requested for subscription individually. In this case all the subscription notifications would each contain the SSC Method Reply to a single subscribed address.

Alternatively, the SSC Server MAY bundle subscription notifications which happen to be sent at the same time into a single notification. The SSC Client MUST be able to handle a bundled notification if it requests multiple subscriptions in a single request, but it MUST NOT rely on the SSC Server bundling the notifications.

In any case the SSC Server SHOULD NOT bundle notification causes, meaning that the SSC Server SHOULD NOT send any subscription notifications for addresses in a bundle with notifications to other addresses, if they would not be sent if all subscriptions had been requested individually.

If some of the SSC addresses in a subscription request must be rejected with errors, whereas other subscriptions succeed, then the SSC Server MAY reject the request completely with an error reply detailing all the failed addresses. If possible, the SSC Server SHOULD instead execute the successful subscriptions and only reject the erroneous ones. This MUST result in a successful reply message to the subscription request, with the reply value including only the successful addresses. In this case the SSC Error state MUST be set to “210 – Partial Success”, and MAY be accompanied by a parameter named “failed_addresses” with an Array of Address trees composed of all the failed Method Addresses (erroneous Addresses replaced by {}), in bundled or unbundled representation. The value of the Address in the Address Tree SHOULD be set to the SSC Error Code relating to the failure of the specific Address. See also the transaction example.

The SSC Server MAY also send an SSC Error “210 – Partial Success” when in fact all of the subscriptions have failed, because the SSC Client receives sufficient information in this Error Reply to work out this fact.

5.4 Supscription request and reply syntax

The SSC Address for subscriptions is `/osc/state/subscribe`.

This SSC Method may be called with a null parameter, which results in an SSC Address tree of all addresses currently subscribed by the SSC Client on the current connection.

The SSC Method also takes a structured parameter, specified as a JSON array.

Each element of the array is an SSC Address Tree specifying the SSC addresses that the SSC Client requests to subscribe. The SSC Address Tree MAY contain Address patterns.

Subscription parameters are specified by embedding them into the Address Tree object as the first JSON object name/value pair with the special name „#" (which can not appear as an Address). The value MUST be a JSON object containing one or more optional subscription parameters by name and value. The subscription parameters are applied for subscribing all SSC Method Addresses in the Address Tree that contains the parameter object. The „#" name/value pair SHOULD be the first item, otherwise the behaviour of the SSC Server MAY depend on the implementation.

An SSC Server that supports subscription MUST be able to interpret a single Address Tree element in the Method Argument array. Multiple Address Trees MAY be supported, or the SSC Server MAY reject them with an SSC Error 414 (request too complex).

The Response to the subscription Request will normally echo the Request, if all subscriptions can be handled successfully. If subscription parameters were requested, then the SSC Server MAY adapt the requested parameters, and MUST send back the adapted parameter values in the Reply. If multiple subscriptions are requested in a single Request, then the SSC Server might find it necessary to adapt subscription parameters differently for different Addresses. In that case, the array in the Reply MAY contain additonal Address trees containing additional adapted parameter objects. The SSC Server MAY also reject the subscription request completely (with SSC Error code 406), or partially (with SSC Error code 210) in such a case.

6. SSC Transport Layer Adaptations

The SSC data format as defined in the previous sections can be transported by different transport protocols, or stored in persistent files. This section specifies what transports are supported, and how the specific features of transport layers shall be applied to transporting SSC Messages.

If an SSC Server supports more than a single transport for SSC, it SHALL behave consistently regardless of the transport used.

6.1 UDP/IP

UDP/IP is the standard transport for all devices with an Ethernet interface or another interface typically used for internet connectivity. All those device MUST implement the UDP/IP transport for SSC.

All devices SHALL implement UDP over IPv6. Support for UDP over IPv4 is OPTIONAL.

One UDP datagram is used to transport one SSC Message. If the SSC Message is really large (e.g., a complete device configuration), IP fragmentation might fail, if a restricted device does not implement IP re-assembly properly. In that case, the SSC Server should break up the message into multiple SSC Method Calls instead. If atomic execution is relevant, SSC time tags may be used.

The UDP port number to used by the SSC Server should normally be discovered by the SSC Client by means of the server discovery protocol. The default port number is 45.

Rationale: No other standard UDP service is expected to use 45. The IANA reservation for a “Message Passing Service” is historic, and SSC is actually passing messages itself. Sennheiser was founded in 1945.

6.2 SSC Server Discovery

Networked devices implement DNS-SD (Apple Bonjour) as discovery protocol.

The DNS Service-Type is specified as „_ssc“.

Because all networked SSC Servers must implement SSC-over-UDP, they MUST all publish a DNS-SD service under „_ssc._udp”. Those servers that additionally support TCP MUST publish another DNS-SD service under „_ssc._tcp”.

The DNS-SD service instance name must be identical to the device name accessible as /device/name. DNS-SD automatic name collision resolution SHOULD be performed, and the resulting name changes MUST be reflected back into /device/name and the persistent device configuration. The renaming rules MAY be tailored to suit product specific requirements.

The DNS-SD service registration includes the port numbers used. SSC Clients SHOULD NOT rely on default ports.

The DNS-SD hostname SHOULD NOT be presented to the user. It may contain a unique identification part (e.g., derived from the device MAC or serial), to avoid name collisions and automatic renaming.

Additional information about the SSC Server may be provided with an DNS-SD TXT-record.

The following properties are currently defined for the TXT record:

- txtvers Version of the TXT record format. Currently „1”.
- version SSC-Version provided by the SSC Server.
- http-uri If this SSC Server offers HTTP(S) transport, the base SSC request URI, including TCP port number.

An SSC Server providing SSC-over-HTTP transport MUST only publish a DNS-SD record as a web server „_http._tcp”, if it provides a web app or configuration page of interest for the human user. The URI published in the HTTP DNS-SD-TXT-record is expected to differ from that for SSC-over-HTTP.

7. SSC Method List

7.1 „/m/beam/elevation“

The metering method returns the elevation of the beam in degree.

- type: Number
- const: false
- writeable: false
- max: 90
- min: 0
- subscr: true

Example SSC get:

```
{„m“:{„beam“:{„elevation“:null}}}
```

Example SSC response:

```
{„m“:{„beam“:{„elevation“:42}}}
```

7.2 „/m/beam/azimuth“

The metering method returns the azimuth of the beam in degree.

- type: Number
- const: false
- writeable: false
- max: 359
- min: 0
- subscr: true

Example SSC subscr:

```
{“osc”:{“state”:{“subscribe”:[ {“m”:{“beam”:{“azimuth”:null}}}]}}}
```

Example SSC get:

```
{„m“:{„beam“:{„azimuth“:null}}}
```

Example SSC response:

```
{„m“:{„beam“:{„azimuth“:231}}}
```

7.3 „/m/in1/peak“

The metering method returns the current detected input peak level.

- type: Number
- const: false
- writeable: false
- units: dB
- max: 0
- min: -90
- subscr: true

Example SSC get:

```
{„m“:{„in1“:{„peak“:null}}}
```

Example SSC response:

```
{„m“:{„in1“:{„peak“:-56}}}
```

7.4 “/device/identification/visual”

If set to true the device will go to identification state. The LEDs will blink for 10 seconds. Parameter resets to false after the timeout of 10 seconds.

- type: Boolean
- const: false
- writeable: true
- subscr: true

Example SSC set:
```
{“device”:{“identification”:{“visual”:true}}}
```
Example SSC get:
```
{“device”:{“identification”:{“visual”:null}}}
```
Example SSC response:
```
{“device”:{“identification”:{“visual”:true}}}
```

7.5 “/device/button/state”

Returns the state of the reset button.

- type: Boolean
- const: false
- writeable: false
- subscr: true

Example SSC get:
```
{“device”:{“button”:{“state”:null}}}
```
Example SSC response:
```
{“device”:{“button”:{“state”:false}}}
```

7.6 “/device/button/label”

Returns the label of the reset button.

- type: String
- const: true
- writeable: false

Example SSC get:
```
{“device”:{“button”:{“label”:null}}}
```
Example SSC response:
```
{“device”:{“button”:{“label”:”Reset”}}}
```

7.7 “/device/button/description”

Description for container device/button.

- type: String
- const: true
- writeable: false

Example SSC get:
```
{“device”:{“button”:{“description”:null}}}
```
Example SSC response:
```
{“device”:{“button”:{“description”:”Reset Button”}}}
```

7.8 “/device/identity/version”

Returns the firmware version of the device.

- type: String
- const: true
- writeable: false

Example SSC get:

```
        {“device”:{“identity”:{“version”:null}}}
```

Example SSC response:

```
        {“device”:{“identity”:{“version”:”1.0.0”}}}
```

7.9 “/device/identity/vendor”

Vendor name of the product.

- type: String
- const: true
- writeable: false

Example SSC get:

```
        {“device”:{“identity”:{“vendor”:null}}}
```

Example SSC response:

```
        {“device”:{“identity”:{“vendor”:”Sennheiser electronic GmbH & Co.
KG”}}}
```

7.10 “/device/identity/serial”

Returns the serial number of the device.

- type: String
- const: true
- writeable: false

Example SSC get:

```
        {“device”:{“identity”:{“serial”:null}}}
```

Example SSC response:

```
        {“device”:{“identity”:{“serial”:”1029100081”}}}
```

7.11 “/device/identity/product”

Returns the identity of the product.

- type: String
- const: true
- writeable: false

Example SSC get:

```
        {“device”:{“identity”:{“product”:null}}}
```

Example SSC response:

```
        {“device”:{“identity”:{“product”:”SLCM2”}}}
```

7.12 “/device/identity/hw_revision”

Returns device hardware revision.

- type: String
- const: true
- writeable: false

Example SSC get:

```
{“device”:{“identity”:{“hw_revision”:null}}}
```

Example SSC response:

```
{“device”:{“identity”:{“hw_revision”:”0”}}}
```

7.13 “/device/update/progress”

Progress of device firmware update.

- type: Number
- const: false
- writeable: false
- max: 100
- min: 0
- subscr: true

Example SSC get:

```
{“device”:{“update”:{“progress”:null}}}
```

Example SSC response:

```
{“device”:{“update”:{“progress”:37}}}
```

7.14 “/device/update/error”

Result of of device firmware update.

- type: String
- const: false
- writeable: false
- subscr: true

Example SSC get:

```
{“device”:{“update”:{“error”:null}}}
```

Example SSC response:

```
{“device”:{“update”:{“error”:”NONE”}}}
```

7.15 “/device/update/enable”

Set to true, to start the firmware update. The device sets to false in case the firmware update ended (after reboot or error).

Abortion by client is not possible.

- type: Boolean
- const: false
- writeable: true
- subscr: true

Example SSC set:

```
{“device”:{“update”:{“enable”:true}}}
```

Example SSC get:

```
{“device”:{“update”:{“enable”:null}}}
```

Example SSC response:

```
{“device”:{“update”:{“enable”:true}}}
```

7.16 “/device/led/custom/color”

Custom LED color during power cycle. The selected color will be shown for 5 seconds.

Activate with `{“device”:{“led”:{“custom”:{“active”:true}}}}`

- type: String
- const: false
- writeable: true
- options: 1. WHITE 2. GREEN 3. BLUE 4. RED 5. YELLOW 6. ORANGE 7. CYAN 8. PINK
- subscr: true

Example SSC set:
```
{“device”:{“led”:{“custom”:{“color”:”CYAN”}}}}
```
Example SSC get:
```
{“device”:{“led”:{“custom”:{“color”:null}}}}
```
Example SSC response:
```
{“device”:{“led”:{“custom”:{“color”:”CYAN”}}}}
```

7.17 “/device/led/custom/active”

Set to “true” to activate custom LED color. Hint: After reboot this will be set back to false.

- type: Boolean
- const: false
- writeable: true
- subscr: true

Example SSC set:
```
{“device”:{“led”:{“custom”:{“active”:true}}}}
```
Example SSC get:
```
{“device”:{“led”:{“custom”:{“active”:null}}}}
```
Example SSC response:
```
{“device”:{“led”:{“custom”:{“active”:true}}}}
```

7.18 “/device/led/mic_mute/color”

User defined mic mute color. The selected color will be shown for 5 seconds.

- type: String
- default: RED
- const: false
- writeable: true
- options: 1. WHITE 2. GREEN 3. BLUE 4. RED 5. YELLOW 6. ORANGE 7. CYAN 8. PINK
- subscr: true

Example SSC set:
```
{“device”:{“led”:{“mic_mute”:{“color”:”ORANGE”}}}}
```
Example SSC get:
```
{“device”:{“led”:{“mic_mute”:{“color”:null}}}}
```
Example SSC response:
```
{“device”:{“led”:{“mic_mute”:{“color”:”ORANGE”}}}}
```

7.19 “/device/led/mic_on/color”

User defined mic on color. The selected color will be shown for 5 seconds.

- type: String
- default: GREEN
- const: false
- writeable: true
- options: 1. WHITE 2. GREEN 3. BLUE 4. RED 5. YELLOW 6. ORANGE 7. CYAN 8. PINK
- subscr: true

Example SSC set:

```
{“device”:{“led”:{“mic_on”:{“color”:”BLUE”}}}}
```

Example SSC get:

```
{“device”:{“led”:{“mic_on”:{“color”:null}}}}
```

Example SSC response:

```
{“device”:{“led”:{“mic_on”:{“color”:”BLUE”}}}}
```

7.20 “/device/led/brightness”

LED brightness in 6 steps. Set to ‘0’ to turn off LEDs.

- type: Number
- const: false
- writeable: true
- max: 5
- min: 0
- subscr: true

Example SSC set:

```
{“device”:{“led”:{“brightness”:5}}}
```

Example SSC get:

```
{“device”:{“led”:{“brightness”:null}}}
```

Example SSC response:

```
{“device”:{“led”:{“brightness”:5}}}
```

7.21 “/device/network/ether/macs”

List of all MAC addresses.

- example:
- type: String
- const: true
- writeable: false
- count: 1

Example SSC get:

```
{“device”:{“network”:{“ether”:{“macs”:null}}}}
```

Example SSC response:

```
{“device”:{“network”:{“ether”:{“macs”:[“00:1B:66:7E:F1:98”]}}}}
```

7.22 “/device/network/ether/interfaces”

List of IPv4 interface names.

- type: String
- const: true
- writeable: false
- count: 1

Example SSC get:

```
{“device”:{“network”:{“ether”:{“interfaces”:null}}}}
```

Example SSC response:

```
{“device”:{“network”:{“ether”:{“interfaces”:[“PoE/Ctrl”]}}}}
```

7.23 “/device/network/ipv4/netmask”

List of current IPv4 netmasks.

- type: String
- default: [“255.255.255.0”]
- const: false
- writeable: false
- count: 1
- subscr: true

Example SSC get:

```
{“device”:{“network”:{“ipv4”:{“netmask”:null}}}}
```

Example SSC response:

```
{“device”:{“network”:{“ipv4”:{“netmask”:[“255.255.0.0”]}}}}
```

7.24 “/device/network/ipv4/manual_netmask”

List of IPv4 netmasks which will be applied by either setting device/network/ipv4/manual_apply to true or device/network/ipv4/auto to false

- type: String
- default: [“255.255.255.0”]
- const: false
- writeable: true
- count: 1
- subscr: true

Example SSC set:

```
{“device”:{“network”:{“ipv4”:{“manual_netmask”:[“255.255.254.0”]}}}}
```

Example SSC get:

```
{“device”:{“network”:{“ipv4”:{“manual_netmask”:null}}}}
```

Example SSC response:

```
{“device”:{“network”:{“ipv4”:{“manual_netmask”:[“255.255.254.0”]}}}}
```

7.25 “/device/network/ipv4/manual_ipaddr”

List of IPv4 addresses which will be applied by either setting device/network/ipv4/manual_apply to true or device/network/ipv4/auto to false

- type: String
- default: [“192.168.178.23”]
- const: false
- writeable: true
- count: 1
- subscr: true

Example SSC set:

```
{“device”:{“network”:{“ipv4”:{“manual_ipaddr”:[“192.168.178.23”]}}}}
```

Example SSC get:

```
{“device”:{“network”:{“ipv4”:{“manual_ipaddr”:null}}}}
```

Example SSC response:

```
{“device”:{“network”:{“ipv4”:{“manual_ipaddr”:[“192.168.178.23”]}}}}
```

7.26 “/device/network/ipv4/manual_gateway”

List of IPv4 gateways which will be applied by either setting device/network/ipv4/manual_apply to true or device/network/ipv4/auto to false

- type: String
- default: [“192.168.178.1”]
- const: false
- writeable: true
- count: 1
- subscr: true

Example SSC set:

```
{“device”:{“network”:{“ipv4”:{“manual_gateway”:[“192.168.178.1”]}}}}
```

Example SSC get:

```
{“device”:{“network”:{“ipv4”:{“manual_gateway”:null}}}}
```

Example SSC response:

```
{“device”:{“network”:{“ipv4”:{“manual_gateway”:[“192.168.178.1”]}}}}
```

7.27 “/device/network/ipv4/ipaddr”

List of current IPv4 addresses.

- type: String
- default: [“192.168.178.23”]
- const: false
- writeable: false
- count: 1
- subscr: true

Example SSC get:

```
{“device”:{“network”:{“ipv4”:{“ipaddr”:null}}}}
```

Example SSC response:

```
{“device”:{“network”:{“ipv4”:{“ipaddr”:[“169.254.10.30”]}}}}
```

7.28 “/device/network/ipv4/interfaces”

List of current IPv4 interfaces corresponding to the list of interface names in device/network/ether/interfaces.

- type: Number
- const: true
- writeable: false
- count: 1
- subscr: true

Example SSC get:

```
{“device”:{“network”:{“ipv4”:{“interfaces”:null}}}}
```

Example SSC response:

```
{“device”:{“network”:{“ipv4”:{“interfaces”:[1]}}}}
```

7.29 “/device/network/ipv4/gateway”

List of current IPv4 default gateways.

- type: String
- default: [“192.168.178.1”]
- const: false
- writeable: false
- count: 1
- subscr: true

Example SSC get:

```
{“device”:{“network”:{“ipv4”:{“gateway”:null}}}}
```

Example SSC response:

```
{“device”:{“network”:{“ipv4”:{“gateway”:[“169.254.1.1”]}}}}
```

7.30 “/device/network/ipv4/auto”

If set to true the corresponding interface in the IPv4 interface list will activate it’s dhcp client. If no dhcp server is present, link-local addressing will be applied within the link-local range. If set to false the “manual_” prefixed settings are taken over to the current settings as long as the “manual_” settings are valid, otherwise the device returns SSC ERROR 406.

- type: Boolean
- const: false
- writeable: true
- count: 1
- subscr: true

Example SSC set:

```
{“device”:{“network”:{“ipv4”:{“auto”:[true]}}}}
```

Example SSC get:

```
{“device”:{“network”:{“ipv4”:{“auto”:null}}}}
```

Example SSC response:

```
{“device”:{“network”:{“ipv4”:{“auto”:[true]}}}}
```

7.31 "/device/network/mdns"

If set to true the SSC service and hostname of the device will be published via the MDNS protocol.

- type: Boolean
- default: true
- const: false
- writeable: true
- subscr: true

Example SSC set:

```
{"device":{"network":{"mdns":true}}}
```

Example SSC get:

```
{"device":{"network":{"mdns":null}}}
```

Example SSC response:

```
{"device":{"network":{"mdns":true}}}
```

7.32 "/device/timeprecision"

The time precision is constantly 1 second.

- type: Number
- const: true
- writeable: false
- max: 1
- min: 1

Example SSC get:

```
{"device":{"timeprecision":null}}
```

Example SSC response:

```
{"device":{"timeprecision":1}}
```

7.33 "/device/time"

The system time can be set/get.

The integer value represents the seconds since January 1, 2000.

Example: `{"device":{"time":582874956}}` will set the date/time to "Thu Jun 21 05:42:36 UTC 2018" - request with `{"device":{"date":null}}`

- type: Number
- const: false
- writeable: true
- min: 0

Example SSC set:

```
{"device":{"time":582874956}}
```

Example SSC get:

```
{"device":{"time":null}}
```

Example SSC response:

```
{"device":{"time":582874956}}
```

7.34 “/device/system”

User settable parameter which can be used freely to store device information.

- type: String
- default: System
- const: false
- writeable: true
- length: 30
- subscr: true

Example SSC set:

```
{“device”:{“system”:”Device”}}
```

Example SSC get:

```
{“device”:{“system”:null}}
```

Example SSC response:

```
{“device”:{“system”:”Device”}}
```

7.35 “/device/position”

This parameter can be used to define the position of the device in the room. Useful if more than one device is in the same room (device/location).

The lenght is limited to 30.

The position information must consist of characters (a-z or A-Z) or digits (0-9) or blanks.

- type: String
- default: over central table
- const: false
- writeable: true
- subscr: true

Example SSC set:

```
{“device”:{“position”:”middle of meeting room”}}
```

Example SSC get:

```
{“device”:{“position”:null}}
```

Example SSC response:

```
{“device”:{“position”:”middle of meeting room”}}
```

7.36 “/device/name”

The user settable name of the device.

The following rules must be applied:

- The lenght is limited to 8 characters.
- The name must start with a letter (a-z or A-Z).
- The name must end with a letter (a-z or A-Z) or a digit (0-9).
- All other symbols may consist of letters, digits or ’-’ or ’_’.
- type: String
- default: SLCM2
- const: false
- writeable: true
- length: 8
- subscr: true

Example SSC set:

```
{“device”:{“name”:”MIC2_A-1”}}
```

Example SSC get:

```
{“device”:{“name”:null}}
```

Example SSC response:

```
{“device”:{“name”:”MIC2_A-1”}}
```

7.37 “/device/location”

This parameter can be used to define in which room and building the device is located.

The following rules must be applied:

- The lenght is limited to 8 characters.
- The name must start with a letter (a-z or A-Z).
- The name must end with a letter (a-z or A-Z) or a digit (0-9).
- All other symbols may consist of letters, digits or ’-’ or ’_’.
- type: String
- default: Room
- const: false
- writeable: true
- length: 8
- subscr: true

Example SSC set:

```
{“device”:{“location”:”ROOM_C31”}}
```

Example SSC get:

```
{“device”:{“location”:null}}
```

Example SSC response:

```
{“device”:{“location”:”ROOM_C31”}}
```

7.38 “/device/language”

The supported language is English (Great Britain).

- type: String
- default: en_GB
- const: true
- writeable: false

Example SSC get:
```
{“device”:{“language”:null}}
```
Example SSC response:
```
{“device”:{“language”:”en_GB”}}
```

7.39 “/device/date”

Date/time stamp of the device - for more details see explanation of “/device/time”.

- type: String
- const: true
- writeable: false

Example SSC get:
```
{“device”:{“date”:null}}
```
Example SSC response:
```
{“device”:{“date”:”Sat Jan 1 00:00:00 UTC 2000”}}
```

7.40 “/device/restore”

FACTORY_DEFAULTS: Reset device to factory defaults - this will also reboot the device!

AUDIO_DEFAULTS: Reset all audio settings to factory defaults - the device will NOT reboot.

- type: String
- const: false
- writeable: true
- options: 1. FACTORY_DEFAULTS 2. AUDIO_DEFAULTS

Example SSC set:
```
{“device”:{“restore”:”FACTORY_DEFAULTS”}}
```
Example SSC get:
```
{“device”:{“restore”:null}}
```
Example SSC response:
```
{“device”:{“restore”:”FACTORY_DEFAULTS”}}
```

7.41 “/device/restart”

Reboots SLCM2

- type: Boolean
- const: false
- writeable: true

Example SSC set:
```
{“device”:{“restart”:true}}
```
Example SSC get:
```
{“device”:{“restart”:null}}
```
Example SSC response:
```
{“device”:{“restart”:true}}
```

7.42 “/interface/version”

Version of the SSC tree.

This includes versioning of the available parameters and their limits.

- type: String
- const: true
- writeable: false

Example SSC get:

```
{“interface”:{“version”:null}}
```

Example SSC response:

```
{“interface”:{“version”:”1.1”}}
```

7.43 “/audio/equalizer/preset”

Activate custom settings for the 7-band graphical equalizer.

OFF: Bypass EQ and ignore “/audio/equalizer/custom”.

CUSTOM: Activate EQ and “audio/equalizer/custom” will be applied.

- type: String
- default: OFF
- const: false
- writeable: true
- options: 1. OFF 2. CUSTOM
- subscr: true

Example SSC set:

```
{“audio”:{“equalizer”:{“preset”:”CUSTOM”}}}
```

Example SSC get:

```
{“audio”:{“equalizer”:{“preset”:null}}}
```

Example SSC response:

```
{“audio”:{“equalizer”:{“preset”:”CUSTOM”}}}
```

7.44 “/audio/equalizer/custom”

Gain settings for the 7-band graphical equalizer.

The cut-off frequencies are: 125, 250, 500, 1000, 2000, 4000, 8000

Filter quality is Q=1.4142

- type: Number
- default: [0,0,0,0,0,0,0]
- const: false
- writeable: true
- count: 7
- units: dB
- max: 8
- min: -8
- inc: 0.5
- subscr: true

Example SSC set:

```
{“audio”:{“equalizer”:{“custom”:[3,6,-3,2,0,-3,-5]}}}
```

Example SSC get:

```
{“audio”:{“equalizer”:{“custom”:null}}}
```

Example SSC response:

```
{“audio”:{“equalizer”:{“custom”:[3,6,-3,2,0,-3,-5]}}}
```

7.45 “/audio/out1/label”

Label that can be found on the housing for connector “Analog Out”.

- type: String
- const: true
- writeable: false

Example SSC get:

```
{“audio”:{“out1”:{“label”:null}}}
```

Example SSC response:

```
{“audio”:{“out1”:{“label”:”Analog Out”}}}
```

7.46 “/audio/out1/desc”

Description for container audio/out1.

- type: String
- const: true
- writeable: false

Example SSC get:

```
{“audio”:{“out1”:{“desc”:null}}}
```

Example SSC response:

```
{“audio”:{“out1”:{“desc”:”Analog audio output”}}}
```

7.47 “/audio/out1/attenuation”

Analogue output attenuation for audio/out1.

- type: Number
- default: 0
- units: dB
- max: 0
- min: -18
- subscr: true

Example SSC get:

```
{“audio”:{“out1”:{“attenuation”:null}}}
```

Example SSC response:

```
{“audio”:{“out1”:{“attenuation”:-10}}}
```

7.48 “/audio/out2/identity/version”

Software version of the Dante interface.

- type: String
- const: true
- writeable: false

Example SSC get:

```
{“audio”:{“out2”:{“identity”:{“version”:null}}}}
```

7.49 “/audio/out2/network/ether/macs”

List of Dante MAC addresses.

- type: String
- const: true
- writeable: false
- count: 2

Example SSC get:

```
{“audio”:{“out2”:{“network”:{“ether”:{“macs”:null}}}}}
```

Example SSC response:

```
{“audio”:{“out2”:{“network”:{“ether”:{“macs”:[“00:1B:66:7E:F1:98”,”0
0:1B:66:7E:F1:99”]}}}}}
```

7.50 “/audio/out2/network/ether/interfaces”

List of Dante IPv4 interface names.

- type: String
- const: true
- writeable: false
- count: 2

Example SSC get:

```
{“audio”:{“out2”:{“network”:{“ether”:{“interfaces”:null}}}}}
```

Example SSC response:

```
{“audio”:{“out2”:{“network”:{“ether”:{“interfaces”:[“00:1B:66:7E:F1:
98”,”00:1B:66:7E:F1:99”]}}}}}
```

7.51 “/audio/out2/network/ipv4/netmask”

List of current IPv4 netmasks of the Dante primary and secondary interfaces.

- type: String
- const: false
- writeable: false
- count: 2
- subscr: true

Example SSC get:

```
{“audio”:{“out2”:{“network”:{“ipv4”:{“netmask”:null}}}}}
```

7.52 “/audio/out2/network/ipv4/ipaddr”

List of current IPv4 addresses of the Dante primary and secondary interfaces.

- type: String
- const: false
- writeable: false
- count: 2
- subscr: true

Example SSC get:

```
{“audio”:{“out2”:{“network”:{“ipv4”:{“ipaddr”:null}}}}}
```

7.53 “/audio/out2/network/ipv4/interfaces”

List of current IPv4 interfaces corresponding to the list of interface names in audio/out2/network/ether/interfaces.

- type: Number
- const: true
- writeable: false
- count: 2
- subscr: true

Example SSC get:

```
{“audio”:{“out2”:{“network”:{“ipv4”:{“interfaces”:null}}}}}
```

Example SSC response:

```
{“audio”:{“out2”:{“network”:{“ipv4”:{“interfaces”:[1,2]}}}}}
```

7.54 “/audio/out2/network/ipv4/gateway”

List of current IPv4 default gateways of the Dante primary and secondary interfaces.

- type: String
- const: false
- writeable: false
- count: 2
- subscr: true

Example SSC get:

```
{“audio”:{“out2”:{“network”:{“ipv4”:{“gateway”:null}}}}}
```

7.55 “/audio/out2/label”

Label that can be found on the housing for connector audio/out2.

- type: String
- const: true
- writeable: false

Example SSC get:

```
{“audio”:{“out2”:{“label”:null}}}
```

Example SSC response:

```
{“audio”:{“out2”:{“label”:”Dante Primary/Secondary”}}}
```

7.56 “/audio/out2/desc”

Description for container audio/out2.

- type: String
- const: true
- writeable: false

Example SSC get:

```
{“audio”:{“out2”:{“desc”:null}}}
```

Example SSC response:

```
{“audio”:{“out2”:{“desc”:”Redundant digital Dante audio output”}}}
```

7.57 “/audio/source_detection/threshold”

Threshold for detecting the speaker depending on the room noise level.

- type: String
- default: normal_room
- options: 1. quiet_room 2. normal_room 3. loud_room
- subscr: true

Example SSC get:

```
{“audio”:{“source_detection”:{“threshold”:null}}}
```

Example SSC response:

```
{“audio”:{“source_detection”:{“threshold”:”quiet_room”}}}
```

7.58 “/audio/mute”

Mute state of the microphone. Set to true to mute the audio outputs.

- type: Boolean
- default: false
- const: false
- writeable: true
- subscr: true

Example SSC set:

```
{“audio”:{“mute”:true}}
```

Example SSC get:

```
{“audio”:{“mute”:null}}
```

Example SSC response:

```
{“audio”:{“mute”:true}}
```

7.59 “/audio/installation_type”

Type of installation of the microphone. The audio signal will be optimized for the choosen installation type.

- flush_mount: The SLCM2 is mounted directly on the ceiling.
- suspended: The SLCM2 is mounted suspended from the ceiling (at least 30 cm).
- type: String
- default: flush_mount
- options: 1. flush_mount 2. suspended
- subscr: true

Example SSC get:

```
{“audio”:{“installation_type”:null}}
```

Example SSC response:

```
{“audio”:{“installation_type”:”flush_mount”}}
```

7.60 “/osc/state/auth/access”

Reflects access rights to SSC methods. “default:/” means default rights from root and lower.

- type: String
- const: false
- writeable: false

Example SSC get:

```
{“osc”:{“state”:{“auth”:{“access”:null}}}}
```

7.61 /osc/state/prettyprint

SSC reply output style is not supported. Returns false.

Example SSC get:

```
{“osc”:{“state”:{“prettyprint”:null}}}
```

7.62 “/osc/state/close”

SSC connection close.

Example SSC get:

```
{“osc”:{“state”:{“close”:null}}}
```

7.63 /osc/state/subscribe

See chapter “SSC subscriptions”.

Example SSC get:

```
{“osc”:{“state”:{“subscribe”:null}}}
```

7.64 /osc/feature/timetag

SSC timed method execution is not supported. Returns false.

Example SSC get:

```
{“osc”:{“feature”:{“timetag”:null}}}
```

7.65 /osc/feature/baseaddr

SSC interactive method address base is not supported. Returns false.

Example SSC get:

```
{“osc”:{“feature”:{“baseaddr”:null}}}
```

7.66 /osc/feature/subscription

SSC subscriptions are supported. Returns true.

Example SSC get:

```
{“osc”:{“feature”:{“subscription”:null}}}
```

7.67 /osc/feature/pattern

SSC message dispatching and pattern matching are supported. Returns “*?”.

Example SSC get:

```
{“osc”:{“feature”:{“pattern”:null}}}
```

7.68 “/osc/limits”

SSC method parameter range reflection.

Example SSC get:

```
{“osc”:{“limits”:null}}
```

7.69 “/osc/schema”

SSC schema reflection.

Example SSC get:

```
{“osc”:{“schema”:null}}
```

7.70 “/osc/version”

SSC protocol version.

Example SSC get:

```
{“osc”:{“version”:null}}
```

7.71 “/osc/xid”

SSC transaction ID.

Example SSC get:

```
{“osc”:{“xid”:null}}
```

7.72 “/osc/ping”

SSC Ping.

Example SSC get:

```
{“osc”:{“ping”:null}}
```

7.73 “/osc/error”

SSC error state.

Example SSC get:

```
{“osc”:{“error”:null}}
```

8. SSC Error List

- 100 : continue
- 102 : processing
- 200 : OK
- 201 : created
- 202 : adapted
- 210 : partial success
- 310 : subscription terminates
- 400 : message not understood
- 401 : authorisation needed
- 403 : forbidden
- 404 : address not found
- 406 : not acceptable
- 408 : request time out
- 409 : conflict
- 410 : gone
- 413 : request too long
- 414 : request too complex
- 416 : requested range not satisfiable
- 422 : unprocessable entity
- 423 : locked
- 424 : failed dependency
- 450 : answer too long
- 454 : parameter address not found
- 500 : internal server error
- 501 : not implemented
- 503 : service unavailable