Sennheiser Sound Control Protocol (SSC)

versatile command, control, and configuration for networked audio systems

Developer‘s guide for Digital 9000

EM 9046

Sennheiser electronic GmbH & Co. KG

Am Labor 1, 30900 Wedemark, Germany, www.sennheiser.com

TI 1247 v1.0

SENNHEISER

Table of Contents

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 Sennheiser 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 division 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.

1. 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.

1.1 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.

2. 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".

2.1 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
SSC Session	association of a specific SSC Client to an SSC Server
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

3. SSC Data Structure Specification

3.1 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 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 a 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 a SSC Method Call of the SSC Method "/audio/out1/level_db" with the argument 5, presumably to set the level to that level, or as an SSC Method Reply message stating the current level.

3.2 JSON Message Transaction Syntax

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

Prefix "`TX:`" indicates a SSC Message that a SSC Client is sending to a SSC Server.

Prefix "`RX:`" indicates a SSC Message that the SSC Server will send back to the Client.

A 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": -100000 }}}
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.

3.3 SSC JSON Message Syntax

3.3.1 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.

3.3.2 SSC Messages

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

A 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 `{ }`.

3.3.3 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" a 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 a SSC Method Reply Message to the originating SSC Client.

A SSC Server's SSC Methods are arranged in a tree structure called a SSC Address Space. The leaves of this tree are the SSC Methods and the branch nodes are called SSC Containers. A 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 a 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. A 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.

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 }}}
```

3.3.4 SSC Message Dispatching and Pattern Matching

When an SSC Server receives an SSC Message, it must invoke the appropriate SSC Methods in its SSC Address Space based on the SSC Message's SSC Address Patterns. This process is called dispatching the SSC Message to the SSC Methods that match its SSC Address Patterns. All the matching SSC Methods are invoked with the same argument data, namely, the SSC Arguments in the SSC Message.

The parts of an SSC Address or an SSC Address Pattern are the successive names of the JSON object members in the SSC Method Call.

A received SSC Message must be dispatched to every SSC Method in the current SSC Address Space whose SSC Address matches the SSC Message's SSC Address Pattern. An SSC Address Pattern matches an SSC Address if:

1. The SSC Address and the SSC Address Pattern contain the same number of parts; and
2. Each part of the SSC Address Pattern matches the corresponding part of the SSC Address.

A part of an SSC Address Pattern matches a part of an SSC Address if every consecutive character in the SSC Address Pattern matches the next consecutive substring of the SSC Address and every character in the SSC Address is matched by something in the SSC Address Pattern. These are the matching rules for characters in the SSC Address Pattern:

1. ? in the SSC Address Pattern matches any single character
2. * in the SSC Address Pattern matches any sequence of zero or more characters
3. A string of characters in square brackets (e.g., [aeiou]) in the SSC Address Pattern matches any character in the string. Inside square brackets, the minus sign (-) and exclamation point (!) have special meanings:
   - Two characters separated by a minus sign indicate the range of characters between the given two in ASCII collating sequence. A minus sign at the end of the string has no special meaning.
   - An exclamation point at the beginning of a bracketed string negates the sense of the list, meaning that the list matches any character not in the list. (An exclamation point anywhere besides the first character after the open bracket has no special meaning.)

4. A comma-separated list of strings enclosed in curly braces (e.g., {foo,bar}) in the SSC Address Pattern matches any of the strings in the list.
5. Any other character in an SSC Address Pattern can match only the same character.

When an SSC Address Pattern is dispatched to multiple SSC Methods, the order in which the matching SSC Methods are invoked is unspecified.

Support for address pattern matching is OPTIONAL for an SSC Server; it MAY be left out in a restricted implementation. If the SSC Server does not support address pattern matching, it MUST treat the special pattern characters like normal characters. An SSC Client can find out whether address patterns are supported by receiving error replies, or by calling the SSC Method /osc/feature/pattern.

3.3.5 SSC Methods addressing array values

An SSC Method MAY have an array of elementary data as value. The value array MAY be empty. The SSC method MUST NOT switch between returning arrays and elementary data as values.

All the elements of the array SHOULD have the same elementary data type. If the SSC Method is reflected in a corresponding /osc/limits Method, then the limits information SHOULD include the count property to describe the array size, and all elements MUST have the same elementary data type (see section "5.1.6 SSC Method parameter range reflection - /osc/limits" p 22).

An SSC Method that accepts an array as SSC Method Argument SHOULD also accept an elementary value, and handle it by silently converting it to an array containg that elementary value as single element. The SSC Method Reply MUST contain the actual array in this case.

3.3.5.1 Accessing complete arrays

It MUST be possible to invoke an array-valued SSC Method in the same way as an elementary-valued Method.

An SSC Method Call with a JSON null value as argument is used to query the current state of the property addressed by the Method. The SSC Method Reply then contains the array value.

An SSC Method Call with a JSON array as Method Argument is used to change the property addressed by the SSC Method.

```
TX: { "presets": { "bank1": { "carriers": null }}}
RX: { "presets": { "bank1": { "carriers": [470000,470400,470800,
                                           471200,471600] }}}
TX: { "presets": { "bank1": { "carriers": [470000,470450,470800,
                                           471250,471600] }}}
RX: { "presets": { "bank1": { "carriers": [470000,470450,470800,
                                           471250,471600] }}}
```

An array-valued SSC Method MAY put additional requirements on the acceptable size of the array value argument. The SSC Method Reply MAY indicate an SSC Error in this case. SSC Error Code 416 ("requested range not satisfiable") MUST be used to indicate errors caused by the requested array value size.

3.3.5.2 Combined query-and-change of complete arrays

If the array-valued argument of an SSC Method Call contains a combination of null and not-null values, the SSC Server MUST interpret this as a query for the current value of the null-valued elements of the array property, combined with a change of the non-null valued elements. The SSC Method Reply MUST then state the current value of the complete array, after applying the requested changes. A restricted SSC Server MAY respond with an SSC Error 414 ("request too complex") if it does not support this kind of request.

This combined query-and-change request allows an SSC client to directly change specific elements of an array-valued SSC Method while saving an extra Method Call to first request the current values of array elements to keep unchanged. The first SSC Method Call of the following SSC Method transaction is only included to show all current values.

```
TX: { "presets": { "bank1": { "carriers": null }}}
RX: { "presets": { "bank1": { "carriers": [470000,470400,470800,
                                           471200,471600] }}}
TX: { "presets": { "bank1": { "carriers": [ null ,470450, null,
                                           471250, null ] }}}
RX: { "presets": { "bank1": { "carriers": [470000,470450,470800,
                                           471250,471600] }}}
```

3.3.5.3 Accessing array ranges

An SSC Server MAY implement an additional SSC Method Argument syntax to facilitate partially querying and changing of array-valued SSC Methods. The OPTIONAL support of this feature is is indicated in the SSC feature method /osc/feature/array_ranges (See also section "5.1.14.5 Support for array range access" p 29).

Array ranges are specified by an JSON object as part of the SSC Method Arguments. This object MAY contain the keys

- index: specifies the zero-based index of the first element of the requested array range
- count: specifies the number of elements of the requested array range

The range specification object MUST be the first element of the JSON array forming the SSC Method Argument of the SSC Method Call.

In Method Calls to query array-valued SSC Methods, the null Method Argument is replaced by an JSON array consisting of the range specification JSON object only. The SSC Method Reply MUST also contain the range specification inside the value array, followed by the values of the array elements in the specified range. The range specification count MUST be equal to the number of value array elements.

```
TX: { "presets": { "bank1": { "carriers": [ {"index":1,"count":3} ]
                                                                  }}}
RX: { "presets": { "bank1": { "carriers": [ {"index":1,"count":3},
                                        470400, 470800, 471200] }}}
```

An SSC Method Call to change an range of an array value also provides a JSON array as the SSC Method Argument, which contains the range specification as the first element, followed by the values for the array elements inside the specified range. The value for the range count MUST be equal to the number of the value array elements.

```
TX: { "presets": { "bank1": { "carriers": [ {"index":1,"count":3},
                      488000, 488400, 488800 ] }}}
RX: { "presets": { "bank1": { "carriers": [ {"index":1,"count":3},
                      488000, 488400, 488800 ] }}}
```

3.3.5.3.1 Special array range specifications

If the properties index and/or count are missing from a range specification they default to the following values:

- index: zero
- count: current size of the array value addressed by the SSC Method

In SSC Method Replies a default array range specification MUST be suppressed to minimise the message size.

So the following transaction is equivalent to "[Accessing complete arrays]":

```
TX: { "presets": { "bank1": { "carriers": [ {} ] }}}
RX: { "presets": { "bank1": { "carriers": [470000,470400,470800,
                                           471200,471600] }}}
```

If the specified value for index is negative, then the array range shall start at an index of array size plus the specified value (e.g., -1 indicates the last array element).

If the specified value for count is negative, then the array range shall contain as many elements as the array size plus the specified value (e.g., -1 indicates all but one array element).

In SSC Method Replies, all values of a range specification MUST be given as positive numbers, even if they were specified with negative values in the SSC Method Call. Thus the last array element can be requested as follows:

```
TX: { "presets": { "bank1": { "carriers": [ {"index":-1,"count":1} ]
                                                                  }}}
RX: { "presets": { "bank1": { "carriers": [ {"index":4,"count":1},
                                              471600 ] }}}
```

Example requesting a negative count. From the total array size of 5, 5 - 2 = 3 values are returned for count=-2.

```
TX: { "presets": { "bank1": { "carriers": [ {"index":1,"count":-2} ]
                                                                  }}}
RX: { "presets": { "bank1": { "carriers": [ {"index":1,"count":3},
                                        470400,470800,471200 ] }}}
```

The current size of the array addressed by the SSC Method can be queried by specifying an empty array range starting at the last element. The index value given in the array range specification of the Method Reply is one less than the array size.

```
TX: { "presets": { "bank1": { "carriers": [ {"index":-1,"count":0} ]
                                                                  }}}
RX: { "presets": { "bank1": { "carriers": [ {"index":4,"count":0} ]
                                                                  }}}
```

3.3.5.3.2 Error handling

If an SSC Method Call queries an array value, and specifies an array range that is not a subrange of the actual array value, then the SSC Server MUST adapt the specified range to fit the actual array range, in the following order:

- adapt the requested array range index to the nearest valid actual array index, i.e., either the requested index, zero, or one less than the actual array size
- adapt the requested array range count to the nearest possible size, i.e., the requested size, or the number of elements starting at the adapted index.

If an SSC Method Call requests to change an array value, and specifies an array range that is not a subrange of the actual array value, then the SSC Method Reply MUST consist of a regular Reply indicating the actual array size combined with an SSC Error 416 ("requested range not satisfiable", see also section "5.1.2 SSC error state - /osc/error" p 19). The actual array size is indicated by an array range specification with size zero and the index of the last array element, e.g.:

```
TX: { "presets": { "bank1": { "carriers": [ {"index":3,"count":2},
                                              488800, 488800 ] }}}
RX: { "osc":{"error":[{"presets":{"bank1":{"carriers":[ 416,
                {"desc":"requested range not satisfiable"}] }}}]},
       "presets":{"bank1":{"carriers":[{"index":4,"count":0}]}}}
```

3.3.6 Temporal Semantics and SSC Time Tags

Per default, the SSC Server shall invoke the SSC Methods addressed by an SSC Message immediately, i.e., as soon as possible after receipt of the message.

An SSC Server may have access to a representation of the correct current absolute time. The optional SSC Method /device/time can be used to query and optionally set the local SSC time used by a device. SSC does not provide a mechanism for clock synchronisation; if an SSC Server utilises a mechanism like NTP or PTP to sync to the absolute time it should handle request to set its SSC time by introducing a local offset from SSC time to the absolute time.

An SSC Message may contain the SSC method /osc/timetag in addition to other methods. In this case, the SSC Time Tag indicates the time when the SSC Server shall execute all of the methods contained in an SSC Message. If the time represented by the SSC Time Tag is before or equal to the current time, the SSC Server should invoke the methods immediately (unless the user has configured the SSC Server to discard messages that arrive too late). Otherwise the SSC Time Tag represents a time in the future, and the SSC Server must store the SSC Message until the specified time and then invoke the appropriate SSC Methods.

Time tags are represented by a JSON number. The integer part of the number specifies the number of seconds since January 1, 2000, and decimal part specifies subsecond precision. Time tags with a value less than 31622400 (corresponding to January 1, 2001) are interpreted as a time offset relative to the current SSC time of the SSC Server.

The actual precision that the SSC Server supports is implementation dependent; especially, it's allowed for an restricted SSC Server to ignore the fractional part of a time tag without raising an error. An SSC Client may enquire the supported time precision by invoking the method /device/timeprecision.

SSC Method Invocations in the same SSC Message are atomic; their corresponding SSC Methods should be invoked in immediate succession as if no other processing took place between the SSC Method invocations.

3.3.7 SSC Sessions

An SSC Session is defined by the association of a specific SSC Client with an SSC Server. The SSC Server MAY keep state information specific to each SSC Client (e.g., state relating to SSC Method subscriptions, or authorisation). If the SSC Server keeps such state, it MUST be coupled to the SSC Session. When the SSC Session terminates, session state SHOULD be cleared (e.g., all SSC Method subscriptions of the client are cancelled).

An SSC Session begins implicitly with the first SSC Method Call that a specific SSC Client sends to an SSC Server.

The SSC Server SHOULD provide the SSC Method /osc/state/close to allow the SSC Client to actively terminate the SSC Session (see "5.1.8 SSC Session termination - /osc/state/close" p 23).

If the underlying transport protocol is based on connections (e.g., TCP/IP), then the SSC Session MUST last for the duration of the connection. The SSC Session MUST be terminated when the connection is closed. The SSC Server MAY send an SSC Method Reply {"osc":{"state":{"close":true}}} before it terminates the SSC Session. The SSC Client MUST also consider the SSC Session as terminated if the connection is closed.

If the underlying transport protocol is not based on connections (e.g., UDP/IP), then the SSC Session MUST last for at least 60 seconds. The SSC Server SHOULD terminate the session automatically 60 seconds after the last succesful SSC Method Call. Each succesful SSC Method Call sent by the SSC Client MUST reset the automatic SSC Session timeout interval. The SSC Client MAY call the SSC Method /osc/ping for this purpose. The SSC Server SHOULD send an SSC Method Reply {"osc":{"state":{"close":true}}} before it terminates the SSC Session. The SSC Client SHOULD also consider the SSC Session as terminated 60 seconds after receiving the last SSC Message from the SSC Server.

The SSC Server SHOULD keep state information that is explicitly specific to the SSC Session in SSC Methods that are based in the SSC Container /osc/state, e.g., information about SSC Method subscriptions (see"4. SSC subscriptions - /osc/state/subscribe" p 17). Additionally, SSC Server state specific for an SSC Session MAY affect the results of calls to other SSC Methods (e.g., the SSC Server might only allow a single SSC Client at a time, or it might provide an SSC Method for Session authorisation, and reject other SSC Method Calls with SSC Error replies until an SSC Client has been authorised).

4. 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 a 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 a 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.

4.1 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

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.

4.2 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.

4.3 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 a 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.

4.4 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 a 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 a SSC Address Tree specifying the SSC addresses that the SSC Client requests to subscribe. The SSC Address Tree MAY contain Address patterns.

A 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 a 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.

5. General SSC Address Schema

Some parts of the SSC address space are reserved by this specification for purposes of meta-protocol information, generic device-independent features, and device capability description.
The reserved parts of the address space are rooted in the addresses:

```
/osc
/device
/internal
```

The addresses and methods rooted in these reserved addresses are described in the following sections. This specification should be revised if additional addresses in the reserved address spaces, or additional reserved address spaces are introduced.

The /internal address space is reserved for internal usage in the SSC Server itself. SSC Clients MUST NOT send any requests addressing methods based in /internal, and SSC Servers MUST NOT implement any externally callable methods.

5.1 SSC Meta Information - /osc

5.1.1 SSC Protocol version - /osc/version

Read-only value. Reports the SSC version implemented in the server.

```
TX: { "osc": { "version": null } }
RX: { "osc": { "version": "1.2" } }
```

5.1.2 SSC error state - /osc/error

Read-only method. Typically this method is not requested actively by the client, but the server sends it as the SSC Method Reply to a faulty SSC Method Call. Simple example:

```
TX: { "out1": { "xlr23": { "gain": 10 } } }
RX: { "osc": { "error": [ { "out1": { "xlr23": [ 404,
                            { "desc": "not found" } ] } } ] } }
```

The error method result MUST contain an array of all the error messages resulting of all method executions in the client message.

The error method result array MUST contain as elements one or more SSC Address Trees corresponding to the method address of each faulty method execution (in the example: "/out1/xlr23", corresponding to the JSON object chain { "out1": { "xlr23": ... } }).

The value of each of the error object chains MUST be an array; this is the actual error message resulting from executing the specified method address. Typeset in **bold** in the example.

The error message MUST contain an integer numeric value, the error code. The error code SHOULD be chosen from the list of error codes detailed below. The SSC Server MAY send different error codes, which then SHOULD be chosen in the same spirit as the canonical error codes.

The error message MAY contain additional information about the error. This will be contained in a JSON object, given as the second item of the error array. Exceptionally this array object has arbitrary properties. The additional information MAY contain a human-readable description of the error; this MUST be sent as a string value for the property named desc (for "description"). The additional error information MAY contain other properties intended for debug or service purposes,

or future protocol extensions. The SSC Client MUST ignore any properties that it does not know.
If the SSC Server sends a non-canonical error message, it SHOULD supply human-readable "desc" information as well, because the SSC Client can't be expected to react in any specific way to an unknown error, other than to relay the description to the user.

The language of any error description MAY depend on the optional /device/language setting.

The following complex example shows how an SSC Message containing three SSC Method calls is answered, where two methods fail and one succeeds:

```
TX: { "out1": { "xlr1": { "mute": true }, "xlr23": { "gain": 3 }},
                                        "out2": { "xlr1": { "gain": 42 }}}
RX: { "osc": { "error": [ { "out1": { "xlr23": [ 404,
      { "desc": "not found" }]}, "out2": { "xlr1": [ 307,
            { "desc": "not just now" }]}} ] },
                  "out1": { "xlr1": { "mute": true }} }
```

If the request message violates the JSON syntax, the complete message cannot reliably be parsed and MUST NOT be partially parsed or executed, so that the SSC Server MUST send an error response (400, "not understood") relating to the complete message, not to any method address. This error result message would look like this:

```
TX: { "out1": { "xlr23": { "ga schnr blabl
RX: { "osc": { "error": [ [ 400, { "desc": "not understood" }]]}}
```

Error method results for successful method executions MUST NOT be sent without being explicitly requested by the client, by querying "/osc/error". Example:

```
TX: { "out1": { "xlr1": { "gain": 17 }},   "osc": { "error": null }}
RX: { "osc": { "error": [ { "out1": { "xlr1": { "gain": [ 202,
    { "desc": "adapted" }]}}} ] },"out1": { "xlr1": { "gain": 15 }}}
```

The error code is a three digit integer, defined in the style of SMTP, FTP or HTTP error codes.

The first digit of the error code defines the class of response. The last two digits do not have any categorization role.

There are 5 values for the first digit:

- 1xx - Informational response - Request received, continuing process.
- 2xx - Success - The action was successfully received, understood, and accepted.
- 3xx - Incomplete - Further action must be taken in order to complete the request.
- 4xx - Client Error - The request contained bad syntax or cannot be fulfilled.
- 5xx - Server Error - The server failed to fulfill an apparently valid request.

A simple SSC Client would only have to look at the first digit of the error code in order to determine what how to deal with the Method Reply.

5.1.2.1 1xx - Informational response

interim status for time-consuming methods.

- 100 continue
- 102 processing

5.1.2.2 2xx – Success

- 200 OK
- 201 Created
- 202 Adapted
- 210 Partial Success

5.1.2.3 3xx – Incomplete

- 310 subscription terminates

5.1.2.4 4xx - Client Error

- 400 message not understood
- 401 authorisation needed
- 403 forbidden
- 404 address not found
- 406 not acceptable (e.g., wrong type for parameter)

- 408 request time out
- 409 conflict
- 410 gone
- 413 request too long
- 414 request too complex
- 416 requested range not satisfiable
- 422 unprocessable entity (error in a complex method parameter)
- 423 locked
- 424 failed dependency
- 450 answer too long
- 454 parameter address not found (e.g., address in a subscription request)

5.1.2.5 5xx - Server Error

- 500 internal server error
- 501 not implemented
- 503 service unavailable

5.1.3 SSC transaction ID - /osc/xid

When an SSC Clients calls the Method /osc/xid, the parameters supplied for the method will be reflected back in the Method Reply of the SSC Server. This can be used by the client to keep track of client-side per-server state.

```
TX: { "osc": { "xid": 1234567, "version": null }}
RX: { "osc": { "xid": 1234567, "version": "1.1" }}
```

See also section "4. SSC subscriptions - /osc/state/subscribe" p 17 for a special application of this Method to subscriptions.

5.1.4 SSC Ping - /osc/ping

When a client invokes the /osc/ping method the server will immediately respond with an /osc/ping response with identical result as invoked. "color:orange">The invoked parameters in an array COULD exceptionally have arbitrary properties.

With no parameters:

```
TX: { "osc": { "ping": null }}
RX: { "osc": { "ping": null }}
```

With some parameters:

```
TX: { "osc": { "ping": [ "abcdefghijklm", 3.14159 ] }}
RX: { "osc": { "ping": [ "abcdefghijklm", 3.14159 ] }}
```

5.1.5 SSC Schema reflection - /osc/schema

The /osc/schema method exists to allow clients to query servers about what address schemes are available on a specific server.

For instance, in our standard example the following Method Call on the top level /out1 address

```
TX: { "osc": { "schema": [ { "out1": null } ] }}
```

would return the following SSC Reply Message which describes the addresses that are contained in the /out1 address one level deep:

```
RX: { "osc": { "schema": [ { "out1": { "xlr1": {}, "xlr2": {} }} ] }
```

An alternative representation of the same SSC Reply in a format that unbundles the SSC Address Tree into an array of SSC Addresses is:

```
RX: { "osc": { "schema": [ { "out1": { "xlr1": {} }},
                                    { "out1": { "xlr2": {} }} ] }}
```

SSC Clients MUST be able to understand both bundled and unbundled Replies.

Note that the responses are empty JSON objects if the address is an SSC Container for more addresses, JSON null if the address is an SSC Method Address.

The method /osc/schema may be called with a null parameter. This is equivalent to querying for the root address schema.

The SSC Client is able to enumerate the complete SSC Address Space of the SSC Server by starting with a query for the address root scheme { "osc": { "schema": null }}, and recursively querying all the SSC Addresses where the replies point to SSC containers.

5.1.6 SSC Method parameter range reflection - /osc/limits

The /osc/limits method allows clients to query what kind of values and what range are accepted by the server in an SSC Method call as parameter values. The response of the request is always a JSON array containing a JSON object describing properties of the addressed SSC Method. These properties MUST be constant during runtime of the device.

The property list is extensible for application-specific features as well as for revised versions of this specification.

Currently defined optional properties are:

- "type": string
  - "Number", "String", "Boolean", or "Container"
- "min": number
  - minimum valid value
- "max": number
  - maximum valid value
- "inc": number
  - recommended user interface increment value
- "length": number
  - maximum length of a string
- "count": number
  - count of array elements that can be expected in responses and that has to be used for settings. -1 represents a not constant or indistinct size (see also section "3.3.5 SSC Methods addressing array values" p 13.
- "subscr": boolean
  - if false then the value can not be subscribed, if true then the value can be subscribed.
- "const": boolean
  - if false then the value can change during runtime, if true then the value is constant. "const":true implies "subscr":false and "writeable":false.
- "writeable": boolean
  - if false then the value can not be set, if true then the value can be changed by command. "writeable":true implies "const": false
- "units": string
  - String describing value units (preferably SI)
- "desc": string
  - descriptive text, meant for display to the user
- "desc_ref": string
  - reference to a not constant description node
- "option": string
  - array of all allowed options for the value
- "option_desc": string
  - array with description text relating to the option values

The language for "units", "description", and "option_desc" MAY depend on /device/language, see also section "7.4.40 /device/language" p 44".

Examples:

```
TX: { "osc": { "limits": [
          { "out1": { "xlr1" : { "level" : null }}} ] }}
RX: { "osc": { "limits": [
          { "out1": { "xlr1" : { "level" : [{ "type": "Number",
                                              "min": -10,
                                              "max": 18,
                                              "inc": 3,
                                              "units": "dB",
                                              "desc": "output level"
                                    }] }}} ] }}

TX: { "osc": { "limits": [ { "main_format": null } ] }}

RX: { "osc": { "limits": [ { "main_format" : [{ "type": "String",
       "desc": "main output mode",
       "option": [ "analogue", "digital" ],
       "option_descr": [ "analogue", "digital AES3" ] }] } ] }}
```

Similar as described for /osc/schema, the SSC Server may format the Method Replies in bundled or unbundled representation of the SSC Addresses, and the SSC Client MUST be able to understand either.

5.1.7 Session-specific SSC Address Space - /osc/state

SSC Methods under the /osc/state Address have results which are specific to the SSC Session between SSC Client and SSC Server. This means that it is possible that different SSC Clients invoke the same SSC Method with different arguments, and the immediate reply as well as the resulting state of the SSC Server will differ for each SSC Client.

This behaviour differs from the normal behaviour of an SSC Server, where the server state is shared between all SSC Clients and connections.

5.1.8 SSC Session termination - /osc/state/close

When an SSC Client calls this SSC Method with a true argument, the SSC Server MUST terminate the SSC Session immediately after the reply has been sent.

In case of an underlying connection-oriented transport like TCP, the SSC Server MUST close the transport-layer connection after the SSC Method Reply has been sent.

When the SSC Session is terminated, the SSC Server clears any state specific to the session, see section "3.3.7 SSC Sessions" p 16.

Example:

```
TX: { "osc": { "state": { "close": true }}}
RX: { "osc": { "state": { "close": true }}}
< Server closes connection >
```

5.1.8.1 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.

If "bw" is greater than 0, then the bandwidth consumed for notification replies is tracked by the SSC Server, and notifications are suppressed when the specified bandwidth would be exceeded. If any notification has been suppressed due to bandwidth limitation, the SSC Server SHOULD send a notification about the actual value of the subscribed address as soon as the bandwidth limitation can be met again. The bandwidth calculation MAY be approximate, and the SSC Client MUST NOT rely on a byte-exact bandwidth limitation.

If multiple subscriptions are requested with a common set of optional parameters, then the optional parameters MUST be interpreted as if all of the requests had been issued separately, each request with the identical set of parameters. Especially, the "bw" parameter imposes the specified bandwidth limit for each subscribed SSC address separately; it is not a summary limit for all of the requested subscriptions.

The SSC Server SHOULD ignore any unknown subscription parameters. Parameter name "internal" is reserved and MUST NOT be used in SSC Client requests.

5.1.8.2 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</s>, default false
- "count": maximum number of notifications to send, 0=unlimited, default 0
- "lifetime": maximum lifetime (s) of the subscription, 0=until SSC session termination, default 0

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.

5.1.8.3 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 section "6.1.8.5 Subscription example transactions" p 25.

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.

5.1.8.4 Subscription 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.

6.1.8.5 Subscription example transactions

Standard case: subscription request, reply and notifications, automatically terminated:

```
TX: { "osc": { "state": { "subscribe": [
                  { "out1": { "xlr2": { "level": null }}} ] }}}

RX: { "osc": { "state": { "subscribe": [
                  { "out1": { "xlr2": { "level": null }}} ] }}}

RX: { "out1": { "xlr2": { "level": 15 }}}
...
RX: { "out1": { "xlr2": { "level": 3 }}}
...
RX: { "out1": { "xlr2": { "level": 9 }}}
...
RX: { "osc": { "error": [ { "out1": { "xlr2": { "level":
       [ 310, { "desc": "subscription terminates" } ] }}}  ] }}
```

Subscribing to multiple addresses in a single request:

```
TX: { "osc": { "state": { "subscribe": [  { "out1": { "xlr\*":
                                          { "level": null }}} ] }}}

RX: { "osc": { "state": { "subscribe": [  { "out1": { "xlr1":
            { "level": null }, "xlr2": { "level": null }}} ] }}}

RX: { "out1": { "xlr1": { "level": 15 },  "xlr2": { "level": 7} }}
...
RX: { "out1": { "xlr1": { "level": 3 }}}
RX: { "out1": { "xlr1": { "level": 5 }}}
...
RX: { "out1": { "xlr2": { "level": 9 }}}
...
```

Subscribing with non-default parameters, partially adapted by the Server:

```
TX: { "osc": { "state": { "subscribe": [ {"#":
            { "min": 96, "max": 50, "lifetime": 3600 },
            "out1": { "xlr2": { "level": null }}} ] }}}

RX: { "osc": { "state": { "subscribe": [ { "#":
            { "min": 100, "max": 50, "lifetime": 600 },
            "out1": { "xlr2": { "level": null }}} ] }}}

RX: { "out1": { "xlr2": { "level": 15 }}}
```

Client cancelling the subscription of the previous example:

```
TX: { "osc": { "state": { "subscribe": [ {
                  "#": { "cancel": true },
                  "out1": { "xlr2": { "level": null }}} ] }}}

RX: { "osc": { "state": { "subscribe": [ {
                  "#": { "cancel": true },
                  "out1": { "xlr2": { "level": null }}} ] }}}
```

Subscribing to multiple addresses, requesting error details, only partially successful:

```
TX: { "osc": { "state": { "subscribe": [
          { "out1": { "xlr1": { "level": null,"nope": null },
              "xlr2": [ "invalid address" ] } } ] },
                  "error": null }}

RX: { "osc": { "state": { "subscribe": [
          { "out1": { "xlr1": { "level": null }}} ] },
              "error": [ "osc": { "state": { "subscribe": [
                  210, {  "desc": "Partial Success",
          "failed_addresses": [ { "osc": { "xlr1": { "nope": 404 }}},
                                  { } ]} ] } } ] } }
```

Subscribing to array elements if array_ranges are supported (see also section "3.3.5.3 Accessing array ranges" p 14"):

```
TX: { "osc": { "state": { "subscribe": [ { "device": { "log":
                          [ {"index":-1, "count":1} ] } } ] } } }

RX: { "osc": { "state": { "subscribe": [ { "device": { "log":
                          [ {"index":-1, "count":1} ] } } ] } } }
...
      {"device":{"log": [{ "index": 1016, "count": 1 },
                          "turbogenerator trip signal blocked"] }}
...
```

```
        {"device":{"log": [{ "index": 1017, "count": 1 },
              "required operating reactivity margin violated"] }}
...
        {"osc":{"error": [{ "device": { "log": [ 310, {"desc":
              "subscription terminates"} ] } }] }}
```

5.1.9 SSC reply output style - /osc/state/prettyprint

An SSC Server MAY support this Method to allow the SSC Client to select a preferred formatting style for all SSC reply messages to be sent on the connection to the SSC Client by the SSC Server. Two styles are defined:

> prettyprint = false: compact representation, no whitespace

> prettyprint = true: formatted by adding whitespace

The exact format of prettyprinted messages depends on the implementation. The SSC Client MUST NOT depend on any specific whitespace decoration. The default style is not specified; the SSC Server MAY choose the default style for each connection independently, e.g., by analysing the first SSC Message sent by the SSC Client; or an UDP SSC Server may use different default output styles on different UDP ports.

Example transaction:

```
TX: { "osc": { "state": { "prettyprint": false }}}
RX: { "osc":{"state":{"prettyprint":false}}}
TX: { "device": { "name": null }}
RX: { "device":{"name":"example device"}}
TX: { "osc": { "state": { "prettyprint": true }}}
RX: { "osc": { "state": { "prettyprint": true }}}
TX: { "device": { "name": null }}
RX: { "device": { "name": "example device" }}
```

5.1.10 SSC interactive method address base - /osc/state/baseaddr

This is an OPTIONAL Method. It helps to explore a device in a truely interactive manner, and may additionally be used to reduce message lengths by shortening addresses in SSC Messages for applications, where an application monitors only a tiny subset of the address space of a complex device with a deeply-nested address space.

The "baseaddr" must specify an existing, valid address on the SSC Server. It is automatically added by the SSC Server to the beginning of any SSC Address in SSC Method calls by the client. It is automatically stripped from replies by the SSC Server to the Client. The base address has effect on messages send over the specific transport-layer client connection.

If the SSC Server can't match an address in an SSC Method call to an address by prepending the base address, it additionally tries to match it as an absolute address. In this way, especially the / osc-address space can be accessed again to reset the base address.

Setting the base address to non-empty fails if the targeted address contains a method or container with the partial address "osc".

Example:

```
TX: { "osc": { "state": {"baseaddr":[{ "out1": {"xlr2": null}}] }}
RX: { "osc": { "state": {"baseaddr":[{ "out1": {"xlr2": null}}] }}
TX: { "gain": -10 }
RX: { "gain": -10 }
```

The SSC Client may query the server for the support of this feature by invoking /osc/feature/baseaddr.

5.1.11 SSC timed method execution - /osc/timetag

When this method is called as part of an SSC Message, all the SSC Method Calls contained in the same Message are to be executed by the SSC Server at a time determined by the timetag parameter. Compare section "3.3.6 Temporal Semantics and SSC Time Tags" p 15 .

```
TX: { "osc": { "ping": null, "timetag": 3.0 }}
```

```
... 3 seconds pass ..
RX: { "osc": { "ping": null }}
```

5.1.12 SSC Method time stamps - /osc/timestamp

This method may be called as part of an SSC Message to estimate the communication delay and clock offset between SSC Client and server. Like in NTP or PTP, four timestamps are used, forming an array. The SSC Client sends the method call and provides the array containing the first, client-side timestamp. The SSC Server will add two server-side timestamps in the SSC Method Reply, and finally the client can insert the fourth timestamp upon reception of the Reply Message.

The four timestamps are:

- sending time of the Message from the SSC Client as value of the client /device/time
- reception time at the SSC Server as value of the server /device/time
- sending time of the Reply at the SSC Server, value of server /device/time
- reception time of the Reply at the client, value of client /device/time

Example:

```
TX: { "osc": { "ping": null, "timestamp": [ 396711511.044569 ] }}
RX: { "osc": { "ping": null,
        "timestamp": [ 396711511.044569, 396711500.05,
                       396711500.08, 396711511.644569 ] }}
```

Support for timestamps is optional. If the SSC Server doesn't implement it, it shall omit the timestamp completely from the Reply Message.

5.1.13 SSC Method Authorisation - /osc/tan

In some applications (like conference systems), remote configurability of SSC devices might pose a security risk, because some attacker with access to the LAN might remotely gain access to sensitive audio streams by reconfiguring devices.

Transaction Authorisation Numbers (TANs) are a simple but efficient means to control configuration access without relying on complex cryptographic protocols, which might be too complex for environments like media control systems.

A list of TAN numbers is distributed between the SSC devices in a system. The SSC Client must provide a valid TAN with each SSC Method Call. The SSC Server refuses to execute the method if it finds the TAN to be invalid. Each TAN may only be used once.

How the TAN list is distributed between the SSC devices in a system is out of the scope of this specification.

```
TX: { "osc": { "ping": null, "tan": "1124frvso!" }}
RX: { "osc": { "ping": null }}
TX: { "osc": { "ping": null, "tan": "1124frvso!" }}
RX: { "osc": { "error": [{
       "osc": { "tan": [{"code": 403, "desc": "TAN invalid"}],
          "ping": [{"code": 403, "desc": "forbidden"}] }}] }}
```

5.1.14 SSC protocol feature reflection - /osc/feature

This SSC Address Space is provided to enable the SSC Client to query the SSC Server whether optional protocol features are supported. In the spirit of extensibility, the SSC Server MUST send a reply with a value of false for each feature that is doesn't know about, even if the feature didn´t exist at all when the server was implemented.

5.1.14.1 Support for pattern matching on SSC addresses

A client may query /osc/feature/pattern to inquire whether the SSC Server supports pattern-matching meta-characters in SSC addresses.
The support is described by a string containing one character for each supported matching pattern:

- * matches on complete address parts are supported

- ? matches on partial addresses are supported
- [matches on character ranges are supported

Example:

```
TX: { "osc": { "feature": { "pattern": null }}}
RX: { "osc": { "feature": { "pattern": "*" }}}
```

If the SSC Server does not support address pattern matching at all, it MAY also reply with false.

5.1.14.2 Support for time tags

A client may query /osc/feature/timetag to inquire whether the SSC Server supports timed method execution.

```
TX: { "osc": { "feature": { "timetag": null }}}
RX: { "osc": { "feature": { "timetag": false }}}
```

5.1.14.3 Support for subscription

A client may query /osc/feature/subscription to inquire whether the SSC Server supports SSC subscription.

```
TX: { "osc": { "feature": { "subscription": null }}}
RX: { "osc": { "feature": { "subscription": true }}}
```

5.1.14.4 Support for method base address

A client may query /osc/feature/baseaddr to inquire whether the SSC Server supports to set a method base address.

```
TX: { "osc": { "feature": { "baseaddr": null }}}
RX: { "osc": { "feature": { "baseaddr": false }}}
```

5.1.14.5 Support for array range access

A client may query /osc/feature/array_ranges to inquire whether the SSC Server supports advanced access to array elements.

```
TX: { "osc": { "feature": { "array_ranges": null }}}
RX: { "osc": { "feature": { "array_ranges": true }}}
```

5.2 Generic Device Information and Settings: Address Space - /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.

5.2.1 /device/identity/product

Read-only string. Product identification, should be identical to the designation on the label on the product itself.

5.2.2 /device/identity/version

Read-only string. Product version, e.g., firmware revision.

5.2.3 /device/identity/serial

Read-only string. *Unique* product number, preferrably identical to the number on a product label.

5.2.4 /device/identity/vendor

Read-only string: often "Sennheiser".

5.2.5 /device/name

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 it COULD be derived from /device/system or/and e.g. /device/location. This name MUST not be misunderstood as channel name that identifies for example an audio link like rx1/name but it is to understand as device identification in a network environment.

If the device is networked, this name shall be used as the name for device discovery (see also section "6.3 SSC Server Discovery" p 34). If device discovery automatically renames the device to resolve naming conflicts, this should be reflected in this property as well as in the display of the device.

Note that the maximum lenght of /device/name which may be represented in the limits (see also section "5.1.6 SSC Method parameter range reflection - /osc/limits" p 22) is dependent on the device discovery concerning unicode length and the name collision resolution.

5.2.6 /device/system

User-settable persistent system name.

This determines the logical system that this device is a part of. It's intended to help the customer to logically separate devices which form different functional systems, but are accessed by means of the same communication medium.

5.2.7 /device/time

Current absolute clock value of the device. Units are seconds, potentially fractional, counting the seconds from 2000-01-01T00:00:00+0000 UTC. The time can be changed by using this address, unless the server is synchronised to absolute time externally, e.g., by means of NTP.
If the device doesn't support absolute clock value, the clock value shall always indicate a time before 2001-01-01T00:00:00+0000, that means the value is greater than 0 and less than 31622400.
If the device doesn't support a clock at all, it shall always return the value 0.

5.2.8 /device/timeprecision

Read-only value specifying the precision of the clock of the device as used for /device/time, SSC time tags, and SSC time stamps.

If the device doesn't support a clock at all, it shall always return the value 0.

5.2.9 /device/language

User-settable value determining the language to be used for values returned by the server which are meant to be displayed to the user. Examples are /osc/error/desc, /osc/limits/.../desc, /osc/limits/.../option_desc.

An SSC Client can determine the possible language options by querying /osc/limits/device/language.

Languages are encoded with 2-3-letter-codes as per the locale convention, e.g. "de", "de_DE". Default language is British English, "en_GB".

Support for languages is optional. Restricted SSC Servers may omit description texts completely; they should return an empty string for /device/language. Servers offering only one fixed language should return that for /device/language, and refuse attempts to change it.

5.2.10 /device/network

This address space allows remote configuration of network settings. Devices without network connectivity don't need to implement the address space. Devices that don't support IPv4 should not

implement the IPv4 address space.

5.2.10.1 /device/network/ether/interfaces

Read-only array containing a list of all the user-relevant ethernet interface of the device.

The names SHALL match the user-readable labeling of the connectors of the device. If the physical port on the device do not carry a textual label, then the textual designation in the user manual of the product SHALL be used.

Internal interfaces which are not accessible to the user MUST NOT be listed here. All accessible physical ports MUST be listed here, even if they all can only be used in a shared configuration, e.g., if they are connected to an internal switch.

5.2.10.2 /device/network/ether/macs

Read-only array containing a list of the MAC addresses of all the user-relevant ethernet interface of the device. The order of the list matches /device/network/ether/interfaces.
The MAC addresses are specified as strings in standard hex-colon-notation.

5.2.10.3 /device/network/ipv4

Address Space for IPv4-specific settings.

These generic methods only relate to the most common case of a device with only a single network interface, or when all of the physical interfaces operate in a bridged configuration, so that only one set of IPv4 network settings is necessary.
More complex devices using different IPv4 addresses on different sets of physical interfaces SHOULD utilise this address space for the main remote control connection, and provide additional address spaces for the remaining interface sets; e.g., /device/network/ipv4-dante.

5.2.10.4 /device/network/ipv4/interfaces

Read-only array relating to /device/network/ether/interfaces. The array contains numbers indexing into the array of physical interface names, and thus also into the array of interface MAC addresses at /device/network/ether/macs.
The interface index array MUST contain the indices of all physical interfaces which share the IPv4 configuration detailed by the following methods.

5.2.10.5 /device/network/ipv4/auto

Boolean value that indicates whether IPv4 shall be configured automatically by means of DHCP and ZeroConf (Auto-IP). If this value is set to true, all the following properties are read-only. true is the default.

5.2.10.6 /device/network/ipv4/ipaddr

Current IPv4 address of the device as a string in standard dot-decimal notation.

5.2.10.7 /device/network/ipv4/netmask

Current IPv4 netmask of the device as a string in standard dot-decimal notation.

5.2.10.8 /device/network/ipv4/gateway

Current IPv4 gateway of the device as a string in standard dot-decimal notation, "0.0.0.0" if no gateway is available.

5.2.10.9 /device/network/ipv6

Address Space for IPv6-specific settings. Like /device/network/ipv4, only the most common case of a single physical interface is specified here. IPv6 SHALL be applied in full autoconfiguration mode only, so that all IPv6 specific method are read-only.

5.2.10.10 /device/network/ipv6/interfaces

Read-only array relating to /device/network/ether/interfaces. The array contains numbers indexing into the array of physical interface names, and thus also into the array of interface MAC addresses at /device/network/ether/macs.
The interface index array MUST contain the indices of all physical interfaces which share the IPv6 configuration accessible by the following methods.

5.2.10.11 /device/network/ipv6/ipaddr

Read-only array of all the IPv6 addresses used on the specified physical interfaces in standard string notation.
The interface specifiers for link-local IPv6 addresses MUST be stripped from the address because they have only internal relevance. An SSC Client application MUST add the interface specifier of the interface local to the client to link-local IPv6 addresses, so that they may be copied and pasted on the client side.

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. An alternative port number is 6970.

- 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. An alternative port number is useful for situations where the default port cannot be used.

6.2 TCP/IP

Support for transporting SSC Messages over TCP/IP is OPTIONAL. An SSC device choosing to support TCP/IP SHOULD support the same set of IP versions for TCP as well as for UDP.

One important application for TCP/IP is to integrate SSC devices into media control systems. In these systems ease of use of the protocol is of special relevance.

TCP/IP is a byte-stream based transport. Therefore it is necessary to define a way of fragmenting the stream into SSC Messages.

The following two-character sequences are recognised as an SSC Message separator:

- ASCII carriage return (13) – newline (10)
- ASCII newline (10) – newline (10)

- Rationale: An unescaped newline cannot appear in the data of a legal SSC Message. The newline character supports interactive use of SSC. Newline alone as single separator character would prevent formatting a large SSC Message over multiple lines (important to store SSC Messages in readable or editable text files).

To support interactive use, SSC Servers providing TCP transport MAY implement the SSC base address feature. They MAY also support the relaxed SSC parser as specified in the appendix.

SSC Messages containing time-critical commands should be pushed (TCP flag PSH) through the TCP stack for minimum latency, after writing the Message separator sequence.

The TCP port number to be used by the SSC Server is expected to be discovered by the client by means of the server discovery protocol. The default port number is 45.

- Rationale: Same as UDP.

6.3 SSC Server Discovery

If IP network devices implement a discovery protocol then it MUST be DNS-SD (Apple Bonjour). The discovery service CAN optionally be enabled or disabled by the application.

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 a 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.
- txtvers: Version of the TXT record format. Currently "2".
- sscvers: SSC-Version provided by the SSC Server.
- model: Model name
- id: Unique identification of the device, derived from the device MACs or serial.
- links: URI(s) of other related services.
- links.sub: URI(s) of other related services in the same device.
- links.http: If this SSC Server offers HTTP(S) transport, the base SSC request URI, including TCP port number.
- info: Optional Information, e.g. "info=simulation". which is not to misunderstand as indicator for device or communication properties but for debugging.

7. Developer's Guide for EM 9046

This chapter describes in detail how a developer should use the SSC interface as implemented for the EM 9046.

7.1 Transport layers

The EM9046 provides SSC service over

- TCP/IPv6 port 45
- UDP/IPv6 port 45
- TCP/IPv4 port 45
- UDP/IPv4 port 45

SSC Clients can choose the transport best suited for the specific application. Metering (SSC Container /m) is best run over UDP, whereas TCP is a better fit for command and control because of the inherent reliability guarantees of TCP.

The EM9046 supports 32 simultaneous SSC Sessions in total, for any combination of transports. Additional attempts to open SSC Sessions are rejected with SSC error 503 ("service not available").

7.2 SSC features

The EM9046 SSC Server supports

- SSC address patterns (section "3.3.4 SSC Message Dispatching and Pattern Matching" p 12)
- accessing array ranges (section "3.3.5 SSC Methods addressing array values" p 13)
- sessions (section "3.3.7 SSC Sessions" p 16)
- subscriptions to all methods (section "4. SSC subscriptions - /osc/state/subscribe" p 17)

Optional subscription parameters (min, max, bw, count, lifetime) are not supported. The lifetime of subscriptions is controlled by the SSC Session, i.e., subscriptions last for the duration of a TCP connection, or while the SSC client sends any SSC request (like /osc/ping) every 60 seconds over UDP.

7.3 Variable SSC address space

The EM9046 can be configured with different modules. Most SSC Methods of the EM9046 are tied to specific modules, thus they are only available for the mounted modules.

All the top-level SSC containers are always present. Top-level containers relating to an unavailable module don't contain any SSC methods. The following containers depend on modules mounted in the respective slots:

- /rx1 (Slot RXD1)
- /rx2 (Slot RXD2)
- /rx3 (Slot RXD3)
- /rx4 (Slot RXD4)
- /rx5 (Slot RXD5)
- /rx6 (Slot RXD6)
- /rx7 (Slot RXD7)
- /rx8 (Slot RXD8)
- /audio1 (Slot OUT1)
- /audio2 (Slot OUT2)
- /audio3 (Slot MAN)

To detect the hardware configuration of an EM9046 device, the following SSC query can be used. The * pattern matches existing SSC addresses only, therefore the reply contains the product types of all mounted modules. Example for an EM9046 with 4 receiver channels and one audio module:

```
TX: { "*": { "identity": { "product": null } } }
```

```
RX: { "rx2": { "identity": { "product": "EM9046DRX" } },
      "rx6": { "identity": { "product": "EM9046DRX" } },
      "rx7": { "identity": { "product": "EM9046DRX" } },
      "rx8": { "identity": { "product": "EM9046DRX" } },
      "audio1": { "identity": { "product": "EM9046DAO" } },
      "device": { "identity": { "product": "EM9046" } }}
```

Metering data in container /m and transmitter information in container /mates depends on the hardware configuration, too.

The SSC method tree of /audio1 and /audio2 differs depending on the type of the audio output module.

For more details, see sections "7.6 /audio* - audio output modules" p 50, "7.7 /m - metering data" p 53, "7.8 /mates - detachable system components" p 55.

7.4 /device - general EM9046 functionality

The EM9046 has two LAN ports, labeled "LAN UP" and "LAN DOWN". Accordingly, all the network configuration methods exhibit arrays of size 2 for values. In current software releases, the EM9046 operates both LAN ports in a bridged configuration. Therefore, the two entries of the arrays are equal. To change network settings, equivalent arrays should be written.

In the following list, enumerated method names are subsumed for clearness.
Container /device/carrier_scan contains 8 subcontainers named carrier_range1 to carrier_range8, where 1-8 correspond to the separate antenna booster ranges.
In the list this is abbreviated as /device/carrier_scan/carrier_range*, where the asterisk represents a digit.

SSC methods that relate to the antenna boosters connected to RF IN A and RF IN B, like /device/carrier_ranges/active, transparently access the antenna boosters of the first EM9046 in an RF-IN daisy-chain (or "cascade") configuration. This also means that changes to such settings may affect all EM9046 devices in the cascade group.

7.4.1 /device/identity/product

Product type: "EM9046"

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

7.4.2 /device/identity/vendor

Vendor name of the product: "Sennheiser electronic GmbH & Co. KG"

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

7.4.3 /device/identity/version

Application version, e.g. "4_0_0_12916"

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

7.4.4 /device/identity/serial

Serial number.

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

7.4.5 /device/name

Network name for identification including discovery.
EM9046 names are always 8 characters long, shorter names are filled with spaces.
Only characters from the set of uppercase letters A-Z, digits 0-9, and $()_~ are allowed.

- limits
  - type: String
  - const: false
  - writeable: true
  - length: 8
  - subscr: true

7.4.6 /device/system

Name of system of related devices. EM9046 sets this to the list of device names of the RF-IN cascade group (names separated by commas). Currently, the user cannot change it.
Example: "JOHN , PAUL , GEORGE , RINGO ".

- limits
  - type: String
  - const: false
  - writeable: false
  - length: 40
  - subscr: true

7.4.7 /device/network/interfaces

List of network interface names: ["LAN UP", "LAN DOWN"].

- limits
  - type: String
  - const: true
  - writeable: false
  - length: 2
  - subscr: true

7.4.8 /device/network/ether/macs

List of all MAC Adresses, e.g. ["00:1b:66:7a:c2:b7", "00:1b:66:7a:c2:b8"].

- limits
  - type: String
  - const: true
  - writeable: false
  - length: 2
  - subscr: true

7.4.9 /device/network/ipv4/netmask

List of current IPv4 netmasks corresponding to list of IPv4 interfaces.

- limits
  - type: String
  - const: false
  - writeable: false
  - length: 2
  - subscr: true

7.4.10 /device/network/ipv4/manual_netmask

List of manual IPv4 netmasks corresponding to list of IPv4 interfaces.

- limits
  - type: String
  - const: false
  - writeable: true
  - length: 2
  - subscr: true

7.4.11 /device/network/ipv4/manual_ipaddr

List of manual IPv4 addresses corresponding list of IPv4 interfaces.

- limits
  - type: String
  - const: false
  - writeable: true
  - length: 2
  - subscr: true

7.4.12 /device/network/ipv4/manual_gateway

List of manual IPv4 gateways corresponding list of IPv4 interface indices.

- limits
  - type: String
  - const: false
  - writeable: true
  - length: 2
  - subscr: true

7.4.13 /device/network/ipv4/manual

List of IP settings modes. If false, then the current IP settings are controlled automatically e.g. by DHCP or ZeroConf. If true, then the manual_* settings are taken over to the current settings as long as the manual_* settings are valid, otherwise the device returns SSC ERROR 406.

- limits
  - type: Boolean
  - const: false
  - writeable: true
  - length: 2
  - subscr: true

7.4.14 /device/network/ipv4/ipaddr

List of current IPv4 addresses corresponding list of IPv4 interface names
 * example: ["10.27.1.46", "10.27.1.46"].

- limits
  - type: String
  - const: false
  - writeable: false
  - length: 2
  - subscr: true

7.4.15 /device/network/ipv4/gateway

List of current IPv4 gateways corresponding list of IPv4 interface names.

- limits
  - type: String
  - const: false
  - writeable: false
  - length: 2
  - subscr: true

7.4.16 /device/network/ipv6/ipaddr

List of current IPv6 addresses.

- limits
  - type: String
  - const: false
  - writeable: false
  - length: 2
  - subscr: true

7.4.17 /device/auth/exclusive

Cooperative lock for exclusive device access for network clients. An SSC Client may write some unique information (e.g., its IP address), and another SSC Client would be able to check that the lock is already taken. The SSC Server does not enforce exclusive access by itself. The lock must be refreshed every 60 seconds.
The Wireless Systems Manager respects the lock. It writes its IPv4 address to claim the lock.

- limits
  - type: String
  - const: false
  - writeable: true
  - length: 16
  - subscr: true

7.4.18 /device/carrier_ranges/min_carrier_frequencies

Antenna booster range minimum frequencies

- limits
  - type: Number
  - count: 8
  - const: false
  - writeable: false
  - units: kHz
  - subscr: true

7.4.19 /device/carrier_ranges/max_carrier_frequencies

Antenna booster range maximum frequencies

- limits
  - type: Number
  - count: 8
  - const: false
  - writeable: false
  - units: kHz
  - subscr: true

7.4.20 /device/carrier_ranges/labels

Antenna booster range designations, e.g. ["B1","B2","B3","B4","B5","B6","B7","B8"]

- limits
  - type: string
  - count: 8
  - const: false
  - writeable: false
  - subscr: true

7.4.21 /device/carrier_ranges/active

Active antenna booster range index 0-7, -1 if no booster connected.

- limits
  - type: Number
  - const: false
  - writeable: true
  - subscr: true

7.4.22 /device/carrier_scan/carrier_range*/rssi_a

Array of received signal strength levels of a carrier scan of antenna booster RF IN A.

- limits
  - type: Number
  - const: false
  - writeable: false
  - count: 960
  - units: dBm
  - max: 0
  - min: –127.5
  - subscr: true

7.4.23 /device/carrier_scan/carrier_range*/rssi_b

Array of received signal strength levels of a carrier scan of antenna booster RF IN B.

- limits
  - type: Number
  - const: false
  - writeable: false
  - count: 960
  - units: dBm
  - max: 0
  - min: –127.5
  - subscr: true

7.4.24 /device/carrier_scan/carrier_range*/control

Write true to start, false to stop range scan.
Read: true if scan is running, false otherwise.

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

7.4.25 /device/carrier_scan/carrier_range*/carrier_frequencies

Array of carrier scan frequencies.

- limits
  - type: Number
  - const: false
  - writeable: false
  - count: 960
  - units: kHz
  - max: 831000
  - min: 470000
  - inc: 25
  - subscr: true

7.4.26 /device/carrier_scan/progress

All-ranges carrier scan progress indicator.
Subscribe to monitor scan progress.

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

7.4.27 /device/presets/bank1/labels

Labels of preset frequencies for /rx*/preset, e.g. ["A3.1", "A3.2", "A3.3", ...]
Depends on the connected antenna boosters and the selected booster range.

- type: String
- const: false
- writeable: false
- count: 0 or 40
- subscr: true

7.4.28 /device/presets/bank1/carrier_frequencies

Preset frequencies selectable for /rx*/preset, e.g. [470200, 470600, 471000, ...]
Depends on the connected antenna boosters and the selected booster range.

- limits
  - type: Number
  - const: false
  - writeable: false
  - count: 0 or 40

- units: kHz
- max: 831000
- min: 470000
- inc: 25
- subscr: true

7.4.29 /device/wordclock/status

Wordclock state description, e.g., "external: 48.0kHz".

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

7.4.30 /device/wordclock/mode

Wordclock setting.

- limits
  - type: String
  - const: false
  - writeable: true
  - options:
    1. internal 44.1kHz
    2. internal 48.0kHz
    3. internal 88.2kHz
    4. internal 96.0kHz
    5. external
    6. MAN (only if EM9046DAN module is present)
  - subscr: true

7.4.31 /device/wordclock/frequency

Wordclock frequency (as used by device).

- limits
  - type: Number
  - const: false
  - writeable: false
  - units: kHz
  - max: 100000
  - min: 40000
  - subscr: true

7.4.32 /device/timezone_offset

Offset of timezone relative to UTC, in seconds.

- limits
  - type: Number
  - const: false
  - writeable: false
  - max: 50400
  - min: −43200
  - subscr: true

7.4.33 /device/timezone_name

Time zone setting.

- limits

- type: String
- const: false
- writeable: true
- options: 70-80 names depending on software release
- subscr: true

7.4.34 /device/timesync_ntp

Enable automatic network time synchronisation (NTP).

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

7.4.35 /device/timeprecision

Time precision (0.1s)

- limits
  - type: Number
  - const: true
  - writeable: false
  - units: s
  - subscr: true

7.4.36 /device/time

Time in millenial seconds, based on UTC.

- limits
  - type: Number
  - const: false
  - writeable: true
  - units: s since 2000-01-01T00:00:00.0Z
  - subscr: true

7.4.37 /device/restore

Write the string "FACTORY_DEFAULTS" to reset the EM9046 configuration.
Reads as "".

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

7.4.38 /device/messages

Array of log messages.

- limits
  - type: String
  - const: false
  - writeable: false
  - count: 1000
  - subscr: true

7.4.39 /device/errors

Array of active error messages.

- limits
  - type: String
  - const: false
  - writeable: false
  - count: 10
  - subscr: true

7.4.40 /device/language

The language/location setting; currently always en_GB.

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

7.5 /rx* - receiver channels

These SSC containers represent the up to 8 receiver channels of the EM9046. Each /rx* container stands for the receiver module in the corresponding slot of the EM9046.

In the method listing, the enumerated method names are subsumed as /rx*. The asterisk represents a digit between 1 and 8.

7.5.1 RF carrier frequency presets

The EM9046 has a common carrier frequency preset bank for all receiver channels.
These carrier frequency presets depend on the connected antenna boosters, and on the selected booster frequency range. The common preset bank is represented at /device/presets/bank1.

Each receiver channel additionally provides an individually configurable preset bank, containing a single preset called U. This is represented at /rx*/presets/bank1.

The SSC containers representing the preset banks contain the array-valued methods labels and carrier_frequencies. These contain the display names of the presets and their corresponding frequency values.

Method /rx*/preset represent the preset setting for a receiver channel. It contains the SSC address of the selected preset, e.g. "/device/presets/bank1/carrier_frequencies[5]", or "presets/bank1/carrier_frequencies[0]" (SSC path relative to same /rx* container). To get the actual preset frequency and label, the indicated SSC address must be queried.

The actual carrier frequency configured by a preset is reflected in the method /rx*/carrier_frequency. If the carrier frequency is changed by directly setting this method, the preset setting changes to the U preset, "presets/bank1/carrier_frequencies[0]".

7.5.2 /rx*/label

Slot label, e.g. "RX8"

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

7.5.3 /rx*/identity/product

Product type, e.g. "EM9046DRX"

- limits

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

7.5.4 /rx*/name

Channel name.

- limits
  - type: String
  - const: false
  - writeable: false
  - length: 8
  - subscr: true

7.5.5 /rx*/carrier_frequency

Channel carrier frequency.

- limits
  - type: Number
  - const: false
  - writeable: true
  - units: kHz
  - max: 831000
  - min: 470000
  - inc: 25
  - subscr: true

7.5.6 /rx*/preset

Channel carrier preset. SSC address of chosen preset carrier frequency.
Either "presets/bank1/carrier_frequencies[0]", or "/device/presets/bank1/carrier_frequencies[n]".

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

7.5.7 /rx*/presets/bank1/labels

Labels of preset frequencies ["U"].

- limits
  - type: String
  - const: true
  - writeable: true
  - count: 1
  - subscr: true

7.5.8 /rx*/presets/bank1/carrier_frequencies

Channel-specific carrier "U" preset.

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

- units: kHz
- max: 831000
- min: 470000
- inc: 25
- subscr: true

7.5.9 /rx*/rf_mode

Channel RF modulation mode. Switches automatically to the received signal, and can not be changed.

- limits
  - type: String
  - const: false
  - writeable: false
  - options: , option_desc
    1. HD , high-definition rf modulation
    2. LR , long-range rf modulation
  - subscr: true

7.5.10 /rx*/encryption

Channel encryption mode

- limits
  - type: String
  - const: false
  - writeable: false
  - options: , option_desc
    1. on , encryption enabled
    2. off , encryption disabled
  - subscr: true

7.5.11 /rx*/enable

Channel audio output enabled.

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

7.5.12 /rx*/commandmode

Channel cmd output mode.

- limits
  - type: String
  - const: false
  - writeable: true
  - options: , option_desc
    1. on , cmd output always on
    2. add , cmd output enabled on button press
    3. mute , main output muted on button press
    4. toggle , cmd and main output toggled on button press
  - subscr: true

7.5.13 /rx*/mates

SSC addresses of transmitters and antenna boosters associated to channel, e.g. ["/mates/antenna_booster1", "/mates/tx1"]

- limits
  - type: String
  - count: variable
  - const: false
  - writeable: false
  - subscr: true

7.5.14 /rx*/audio

SSC addresses of audio outputs connected to channel audio, e.g. ["/audio1/out3", "/audio2/out5", "/audio2/out6"]

- limits
  - type: String
  - count: variable
  - const: false
  - writeable: false
  - subscr: true

7.5.15 /rx*/audio_aux

SSC addresses of audio outputs connected to channel aux audio, e.g. ["/audio1/out9", "/audio2/out9"]

- limits
  - type: String
  - count: variable
  - const: true
  - writeable: false
  - subscr: true

7.5.16 /rx*/warnings

- list of warnings related to channel, e.g. ["no signal"]
- limits
  - type: String
  - count: variable
  - const: false
  - writeable: false
  - subscr: true

7.5.17 /rx*/operation/standby

Channel standby mode.

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

7.5.18 /rx*/operation/monitor

Channel monitor mode.

- limits
  - type: Boolean

- const: false
- writeable: true
- subscr: true

7.5.19 /rx*/sync_settings/rf_mode

Channel transmitter RF modulation mode for sync.

- limits
  - type: String
  - const: false
  - writeable: true
  - options: , option_desc
    1. HD , high-definition rf modulation
    2. LR , long-range rf modulation
  - subscr: true

7.5.20 /rx*/sync_settings/name

Channel transmitter name for sync, always equal to /rx*/name.

- limits
  - type: String
  - const: false
  - writeable: false
  - length: 8
  - subscr: true

7.5.21 /rx*/sync_settings/carrier_frequency

Channel transmitter carrier frequency for sync, always equal to /rx*/carrier_frequency.

- limits
  - type: Number
  - const: false
  - writeable: false
  - units: kHz
  - max: 831000
  - min: 470000
  - inc: 25
  - subscr: true

7.5.22 /rx*/sync_settings/lowcut

Transmitter audio low-cut frequency for sync.

- limits
  - type: Number
  - const: false
  - writeable: true
  - units: Hz
  - options:
    1. 30
    2. 60
    3. 80
    4. 100
    5. 120
  - subscr: true

7.5.23 /rx*/sync_settings/lock

Transmitter key-lock setting for sync.

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

7.5.24 /rx*/sync_settings/gain

Transmitter audio gain for sync.

- limits
  - type: Number
  - const: false
  - writeable: true
  - units: dB
  - max: 60
  - min: – 6
  - inc: 3
  - subscr: true

7.5.25 /rx*/sync_settings/encryption

Channel transmitter encryption mode for sync.

- limits
  - type: String
  - const: false
  - writeable: true
  - options: , option_desc
    1. on , encryption enabled
    2. off , encryption disabled
  - subscr: true

7.5.26 /rx*/sync_settings/display

Transmitter display mode for sync.

- limits
  - type: String
  - const: false
  - writeable: true
  - options: , option_desc
    1. name , display name
    2. preset , display preset
    3. frequency , display frequency
  - subscr: true

7.5.27 /rx*/sync_settings/cable_emulation

Channel transmitter cable emulation mode for sync.

- limits
  - type: String
  - const: false
  - writeable: true
  - options: , option_desc
    1. line , no cable emulation
    2. type1 , cable emulation type 1

3. type2 , cable emulation type 2
4. type3 , cable emulation type 3

- subscr: true

7.6 /audio* - audio output modules

The SSC tree for each audio output module depends on the type of the module.
Slots OUT1 and OUT2 are represented by containers /audio1 and /audio2, respectively.
Both slots can contain an analogue output module EM9046AAO or digital output module EM9046DAO.

Slot MAN is represented by container /audio3.
It can contain a network audio module EM9046DAN.

7.6.1 Analogue output module EM9046AAO

The analogue output module EM9046AAO supports 16 output channels, in containers /audio*/out1 to /audio*/out16. The out* containers correspond to the following connectors (as reflected in SSC method /audio*/out*/label):

- out1: XLR 1
- out2: XLR 2
- ...
- out8: XLR 8
- out9: Analogue Multicore 1
- out10: Analogue Multicore 2
- ...
- out16: Analogue Multicore 8

All outputs provide an SSC method level to set the analogue output level. The levels of the multicore outputs out9 to out16 are linked (due to the hardware setup); if one level is changed, the levels for the other 7 outputs follow suit.

7.6.2 Digital output module EM9046DAO

The digital output module EM9046DAO supports 24 output channels, in containers /audio*/out1 to /audio*/out24. The out* containers correspond to the following connectors (as reflected in SSC methods /audio*/out*/label):

- out1: XLR 1/2 a left
- out2: XLR 1/2 a right
- out3: XLR 3/4 a left
- out4: XLR 3/4 a right
- ...
- out7: XLR 7/8 a left
- out8: XLR 7/8 a right
- out9: XLR 1/2 b left
- out10: XLR 1/2 b right
- out11: XLR 3/4 b left
- out12: XLR 3/4 b right
- ...
- out15: XLR 7/8 b left
- out16: XLR 7/8 b right
- out17: Digital Multicore 1
- out18: Digital Multicore 2
- ...
- out24: Digital Multicore 8

7.6.3 Network audio module EM9046DAN

The digital output module EM9046DAN supports 16 network audio channels, in containers /audio3/out1 to /audio3/out16. The out* containers correspond to the following network streams (as reflected in SSC methods /audio3/out*/label):

- out1: main audio stream 1
- out2: main audio stream 2
- ...
- out8: main audio stream 8
- out9: aux audio stream 1
- out10: aux audio stream 2
- ...
- out16: aux audio stream 8

All outputs provide an SSC method name to reflect the configurable network audio stream name. In current firmware, the stream names can only be queried or changed by means of the external network audio management application. The name method always returns the same result as the label method.

7.6.4 Audio crossbar switch

The EM9046 can connect each RX audio channel to any output in hardware. The RX audio is represented by two logical channels, audio and audio_aux. The cmd mode setting of the RX channel in combination with the state of the (optional) CMD button on the RF transmitter determine whether audio is enabled on audio, or audio_aux, or both.
In current EM9046 software the routing is fixed.

Each logical audio channel of each RX module can be routed to any output on any audio output module. All SSC output containers /audio*/out* provide a method inputs, detailing the audio sources connected to the output channel as an array of SSC addresses pointing to the SSC addresses of the audio source (e.g., ["/rx3/audio", "/rx5/audio_aux"]).

Reciprocally, all receiver channel containers /rx* provide the methods /rx*/audio and /rx*/audio_aux. These methods list the SSC addresses of the outputs connected to the channels.

7.6.5 Audio module methods

In the following method list, the enumerated parts of the methods names are subsumed and represented by an asterisk.

In the pattern /audio*, the asterisk is 1, 2, or 3.
In the pattern /audio*/out*, the second asterisk represents a number from 1 to 16, or 1 to 24 for EM9046DAO modules.

7.6.6 /audio*/label

Slot label: "OUT1", "OUT2", or "MAN" (**M**ulti **A**udio **N**etwork).

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

7.6.7 /audio*/identity/product

Product type ("EM9046AAO", "EM9046DAO", or "EM9046DAN").

- limits
  - type: String
  - const: false

- writeable: false
- subscr: true

7.6.8 /audio*/identity/vendor

Vendor name of the product ("Sennheiser electronic GmbH & Co. KG").

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

7.6.9 /audio*/out*/label

Audio output connector label, e.g. "XLR 3"

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

7.6.10 /audio*/out*/inputs

SSC addresses of audio sources for this output, e.g. ["/rx1/audio"]

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

7.6.11 /audio*/out*/level

Audio output gain (for EM9046AAO modules only).

- limits
  - type: Number
  - const: false
  - writeable: true
  - units: dBu
  - max: 18
  - min: -10
  - inc: 1
  - subscr: true

7.6.12 /audio3/out*/name"

Audio output network stream name, e.g. "main audio stream 3" (for EM9046DAN modules only).

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

7.7 /m - metering data

All metering data is bundled in container /m, so that it can be subscribed comprehensively, and without too much overhead. All metering methods yield arrays, where each element represents data from a specific RX channel. The data sources are listed in method /m/sources as SSC addresses. The first element of /m/sources (e.g., /rx3) means that all first elements of the methods /m/rssi_a, /m/rssi_b, etc., correspond to the respective channel (e.g., CH3).

Metering data is always subscribed completely. Subscribing one metering data method (except /m/sources) implicitly subscribes all other methods, too. It is recommended to subscribe to /m/*. Subscription notifications are sent at a rate of 10/sec.

Example:

```
TX: {"osc":{"state":{"subscribe":[{"m":{"*":null}}]}}}
RX: {"osc":{"state":{"subscribe":
      [{"m":{"af_level":null,"divi_a":null,"divi_b":null,
            "rsqi_a":null,"rsqi_b":null,"rssi_a":null,
            "rssi_b":null,"sources":null}}]}}}

RX: {"m":{"sources":["/rx1", "/rx2", "/rx3", "/rx4", "/rx5", "/rx6",
                        "/rx7", "/rx8"]}}

RX: {"m":{"rssi_a":[ -112.0, -111.5, -111.0, -112.5, -113.5, -111.5,
                        -111.0, -113.5 ],
      "rssi_b":[ -112.0, -111.5, -112.5, -112.0, -112.5,
                        -112.0, -112.0, -111.5 ],
      "rsqi_a":[      0,      0,      0,      0,      0,
                        0,      0,      0 ],

      "rsqi_b":[      0,      0,      0,      0,      0,
                        0,      0,      0 ],

      "divi_a":[      0,      0,      0,      0,      0,
                        0,      0,      0 ],

      "divi_b":[      0,      0,      0,      0,      0,
                        0,      0,      0 ],
      "af_level":[  -127.5, -127.5, -127.5, -127.5, -127.5,
                        -127.5, -127.5, -127.5 ]}}

RX: {"m":{"rssi_a":[ -112.0, -111.5, -111.0, -112.5, -113.5, -111.5,
                        -111.0, -113.5 ],

      "rssi_b":[ -112.0, -111.5, -112.5, -112.0, -112.5, -112.0,
                        -112.0, -111.5 ],

      "rsqi_a":[      0,      0,      0,      0,      0,      0,
                        0,      0 ],

      "rsqi_b":[      0,      0,      0,      0,      0,      0,
                        0,      0 ],

      "divi_a":[      0,      0,      0,      0,      0,      0,
                        0,      0 ],

      "divi_b":[      0,      0,      0,      0,      0,      0,
                        0,      0 ],

      "af_level":[  -127.5, -127.5, -127.5, -127.5, -127.5, -127.5,
                        -127.5, -127.5 ]}}
RX: ...
```

7.7.1 /m/sources

SSC addresses of metering sources, e.g. ["/rx1","/rx2","/rx3","/rx4"]

- limits
  - type: String
  - count: 0-8
  - const: true
  - writeable: false
  - subscr: true

7.7.2 /m/rssi_a

RF A signal strength indicator for each metering rxN source.

- limits
  - type: Number
  - count: 0-8
  - const: false
  - writeable: false
  - units: dBm
  - max: 0
  - min: -127.5
  - subscr: true

7.7.3 /m/rssi_b

RF B signal strength indicator for each metering rxN source.

- limits
  - type: Number
  - count: 0-8
  - const: false
  - writeable: false
  - units: dBm
  - max: 0
  - min: -127.5
  - subscr: true

7.7.4 /m/rsqi_a

RF A signal quality indicator for each metering rxN source.

- limits
  - type: Number
  - count: 0-8
  - const: false
  - writeable: false
  - units: %
  - max: 100
  - min: 0
  - subscr: true

7.7.5 /m/rsqi_b

RF B signal quality indicator for each metering rxN source.

- limits
  - type: Number
  - count: 0-8
  - const: false

- writeable: false
- units: %
- max: 100
- min: 0
- subscr: true

7.7.6 /m/divi_a

RF A diversity indicator for each metering rxN source.

- limits
  - type: Number
  - count: 0-8
  - const: false
  - writeable: false
  - units:
  - max: 1
  - min: 0
  - subscr: true

7.7.7 /m/divi_b

RF B diversity indicator for each metering rxN source.

- limits
  - type: Number
  - count: 0-8
  - const: false
  - writeable: false
  - units:
  - max: 1
  - min: 0
  - subscr: true

7.7.8 /m/af_level

Audio level for each metering rxN source.

- limits
  - type: Number
  - count: 0-8
  - const: false
  - writeable: false
  - units: dBfs
  - max: 0
  - min: -127.5
  - subscr: true

7.8 /mates - detachable system components

This SSC container keeps status information about detachable devices:

- antenna boosters
- RF transmitters

If no device is connected, the SSC methods return dummy information.

The set of active devices can be queried from /mates/active.

There are two antenna boosters and up to eight transmitters, subsumed in the method listing as /mates/antenna_booster*, and /mates/tx*. The asterisk represents a digit.

7.8.1 /mates/active

list of SSC relative addresses of active connected devices, e.g. ["tx1", "tx5", "antenna_booster1", "antenna_booster2"]

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

7.8.2 /mates/antenna_booster*/label

Connector label: "RF IN A"

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

7.8.3 /mates/antenna_booster*/identity/product

Product type, e.g. "AB9000 A1-A8" (empty if not connected)

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

7.8.4 /mates/antenna_booster*/identity/vendor

Vendor name of the product.

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

7.8.5 /mates/antenna_booster*/identity/version

Antenna booster SW version, e.g. "12-05-07-09"

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

7.8.6 /mates/tx*/identity/product

Product type, e.g. "SKM9000" (empty if not connected)

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

7.8.7 /mates/tx*/identity/vendor

Vendor name of the product.

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

7.8.8 /mates/tx*/switch1/state

Channel transmitter switch state (true=pressed).

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

7.8.9 /mates/tx*/switch1/label

Channel transmitter switch designation ("CMD" or "").

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

7.8.10 /mates/tx*/rf_mode

Channel transmitter RF modulation mode.

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

7.8.11 /mates/tx*/name

Channel transmitter name.

- limits
  - type: String
  - const: false
  - writeable: false
  - length: 8
  - subscr: true

7.8.12 /mates/tx*/lowcut

Channel transmitter audio low-cut frequency.

- limits
  - type: Number
  - const: false
  - writeable: false
  - units: Hz
  - max: 120
  - min: 30
  - inc: 1
  - subscr: true

7.8.13 /mates/tx*/lock

Channel transmitter key-lock setting.

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

7.8.14 /mates/tx*/gain

Channel transmitter audio gain.

- limits
  - type: Number
  - const: false
  - writeable: false
  - units: dB
  - max: 60
  - min: -6
  - inc: 1
  - subscr: true

7.8.15 /mates/tx*/encryption

Channel transmitter encryption mode.

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

7.8.16 /mates/tx*/display

Channel transmitter display mode.

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

7.8.17 /mates/tx*/carrier_frequency

Channel transmitter carrier frequency.

- limits
  - type: Number
  - const: false
  - writeable: false
  - units: kHz
  - max: 831000
  - min: 470000
  - inc: 25
  - subscr: true

7.8.18 /mates/tx*/cable_emulation

Channel transmitter cable emulation mode.

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

7.8.19 /mates/tx*/bat_state

Channel transmitter battery state, either bat_lifetime or bat_gauge.

- limits
  - type: Number
  - const: false
  - writeable: false
  - subscr: true

7.8.20 /mates/tx*/bat_lifetime

Channel transmitter battery lifetime in seconds.

- limits
  - type: Number
  - const: false
  - writeable: false
  - units: s
  - subscr: true

7.8.21 /mates/tx*/bat_gauge

Channel transmitter battery state, percent of full.

- limits
  - type: Number
  - const: false
  - writeable: false
  - units: %
  - subscr: true

7.8.22 /mates/tx*/bat_bars

Channel transmitter battery state, count of bars (0..3).

- limits
  - type: Number
  - const: false
  - writeable: false
  - subscr: true

7.8.23 /mates/tx*/af_source

Channel transmitter audio source (e.g., "MMD 935").

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

7.9 /osc - SSC features

7.9.1 /osc/state/prettyprint

SSC reply output style is not supported. Returns false.

7.9.2 /osc/state/close

SSC session close.

7.9.3 /osc/state/subscribe

SSC subscriptions.

7.9.4 /osc/feature/array_ranges

SSC array ranges are supported. Returns true.

7.9.5 /osc/feature/timetag

SSC timed method execution is not supported. Returns false.

7.9.6 /osc/feature/baseaddr

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

7.9.7 /osc/feature/subscription

SSC subscriptions are supported. Returns true.

7.9.8 /osc/feature/pattern

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

7.9.9 /osc/limits

SSC method parameter range reflection.

7.9.10 /osc/schema

SSC schema reflection.

7.9.11 /osc/version

SSC protocol version.

7.9.12 /osc/xid

SSC transaction ID.

7.9.13 /osc/ping

SSC Ping.

7.9.14 /osc/error

SSC error state.

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 : request 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

9. Network Audio Monitoring

The EM9046 provides a network audio streaming service for monitoring purposes.

9.1 Audio Streaming Standards

The audio streaming concept is based on open Internet-standards.

It aims to be compatible with Ravenna, omitting the clock synchronisation features (http://ravenna.alcnetworx.com/technology/technology-overview.html).
Following Internet Standards are applied:

- stream management: Real Time Streaming Protocol (RTSP) (RFC 2326)
- stream information and configuration: SDP: Session Description Protocol (RFC 4566)
- stream format: RTP: A Transport Protocol for Real-Time Applications (RFC 3550)
- RTP profile: RTP Profile for Audio and Video Conferences with Minimal Control (RFC 3551)
- RTP payload formats: MIME Type Registration of RTP Payload Formats (RFC 3555)
- specific RTP payload format: audio/L16 (RFC 3555 section 4.1.15)
- supported sample rates: 44100 Hz and 48000 Hz, depending on EM9046 configuration
- channel layout: 8 interleaved channels ordered 1-2-3-4-5-6-7-8

The EM9046 works as a sending-only streaming server. The stream always contains 8 channels, regardless whether the sources (EM9046 receiver modules) are working, receiving audio signals, or are even installed at all. Unused channels are filled with digital silence (zero-valued samples). Likewise, the "cmd" function of the Digital 9000 system is ignored.

The EM9046 supports a single RTP stream and a single RTSP session. When a second client connects to establish a new RTSP session, the first session will be disconnected.

9.2 Additional Standards

- Multicast DNS (RFC 6762)
- DNS-Based Service Discovery (RFC 6763)
- Defining Well-Known Uniform Resource Identifiers (URIs) (RFC 5785)

9.3 Streaming setup

The EM9046 provides the RTSP/RTP service on the RTSP standard port 554/TCP/IPv4.

The RTSP path is "/.well-known/rtsp", conforming to RFC5785. Redirects must be followed.
The stream will be described in the SDP record correctly as audio/L16/rate/8, and no port change is assumed. This means that the channel to play-back has to be selected in the streaming client by means of the respective streaming player API.

An audio streaming client should follow these steps to connect to the EM9046 streaming service:

- establish TCP connection to the RTSP port of an EM9046
  (if this fails, then the EM9046 doesn't support audio streaming)
- establish RTSP session to RTSP path /.well-known/rtsp,
  following any RTSP redirect response (Status 301, Moved Permanent) (ref. RFC5785)
- follow standard RTSP procedure to SETUP the RTP stream.

Alternatively, the client may take these steps:

- query for DNS-SD instances of service type _ssc._udp
- retrieve TXT record of found instances, and inspect property links.sub for a RTSP-URL.
  Example: links.sub=rtsp://192.0.2.1:554/by-name/ch1,2,3,4,5,6,7,8/
- establish RTSP session to the RTSP-URL, following redirects.

9.4 Ravenna Compatibility

The EM9046 provides additional RTSP paths in a way compatible for Ravenna streaming clients.
It doesn‘t publish Ravenna service instances via DNS-SD.