Sennheiser Sound Control Protocol (SSC)

versatile command, control, and configuration
for networked audio systems

Developer's guide for evolution wireless D1

TI 1094 v1.0

Sennheiser electronic GmbH & Co. KG
Am Labor 1, 30900 Wedemark, Germany
www.sennheiser.com

SENNHEISER

Table of Contents

1. Introduction Sennheiser Sound Control Protocol

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

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

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

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

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

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

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

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

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

2. Open Sound Control Overview

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

The OSC specification Version 1.1 is available from the Open Sound Control website at http://www.opensoundcontrol.org/.

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

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

The OSC Schema designed 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 designed in this document.

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

2.1 JavaScript Object Notation Overview

JavaScript Object Notation (JSON) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language, Standard ECMA-262 3rd Edition - December 1999. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Ruby, Python, and many others. These properties make JSON an ideal data-interchange language.

The central website for JSON infomation is http://json.org. JSON is formally specified in RFC 4627 (MIME-type application/json).

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

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

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

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

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

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

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

3. Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP14/RFC 2119, "Key words for use in RFCs to Indicate Requirement Levels".

3.1 Terminology

SSC Message	protocol unit of transmission
SSC Server	device or application that receives SSC messages, and replies to them
SSC Client	device, application or person that sends SSC messages
SSC Container	named entity containing SSC Methods or other Containers
SSC Method	named attribute or action callable on an SSC Server
SSC Address	full name of an 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 an SSC Server
SSC Method Call	SSC Message requesting execution of an SSC Method
SSC Method Arguments	arguments included in an SSC Method Call
SSC Method Reply	SSC Message send by SSC Server as result of a Method Call
binary OSC	the binary OSC encoding as opposed to JSON-based SSC
restricted SSC Server	an SSC Server that doesn't implement some optional parts of this specification

4. SSC Data Structure Specification

4.1 Applying JSON to the OSC device model

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

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

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

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

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

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

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

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

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

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

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

4.2 JSON Message Transaction Syntax

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

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

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

An SSC-Message is written verbatim, enclosed by curly brackets { }.

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

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

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

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

```
TX: { "out1": { "xlr2": { "gain": -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.

4.3 SSC JSON Message Syntax

4.3.1 Elementary data types

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

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

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

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

- string => 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 => boolean: Any non-empty string is true, an empty string is false.
- number => string: a string representation of the number suitable for interpretation by strtod() is used.
- number => boolean: Number zero is false, everything else is true.
- boolean => string: true results in "true", false in an empty string "".
- boolean => number: true is 1, false is 0.

4.3.2 SSC Messages

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

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

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

The length of an SSC Message is variable. If the underlying transport protocol is packet-based, like UDP/IP or ZigBee, then exactly one SSC Message SHOULD be contained in one transport packet. If the underlying transport protocol is a byte-stream, like TCP/IP or a serial link, then SSC Messages MUST be terminated, additionally to the grouping provided by the JSON syntax, with Message Separator Characters specific to the transport. The message separator characters MUST NOT be able to occur unescaped in the non-ignored contents of the packet. Compare section 6.2, page 30.

4.3.3 SSC Addresses

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

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

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

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

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

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

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

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

Example:

- query current gain of XLR2 output of OUT1 module:

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

- change gain of XLR2 output of OUT1 module (note that the server adapts the value):

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

4.3.4 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:

- The SSC Address and the SSC Address Pattern contain the same number of parts; and
- 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:

- in the SSC Address Pattern matches any single character
- * in the SSC Address Pattern matches any sequence of zero or more characters
- 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.)
- 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.
- 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`.

4.3.5 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.

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": "1.1" }}
```

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. 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.

1xx – Informational response

interim status for time-consuming methods.

100 continue

102 processing

2xx – Success

200 OK

201 Created

202 Adapted

210 Partial Success

3xx – Incomplete

310 subscription terminates

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

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)

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 5.1.9 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.

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.

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
- `units` string String describing value units (preferably SI)
- `desc` string descriptive text, meant for display to the user
- `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 section 5.2.9.

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 Connection-specific SSC Address Space - /osc/state

SSC Methods under the `/osc/state` Address have results which are specific to the connection 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 connection close - /osc/state/close

When an SSC Client calls this SSC Method with a `true` argument, the SSC Server MUST close the connection 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. In case of connection-less transport layers like UDP, the SSC Server MUST consider the specific client as gone and clear all state associated with the client. For example, the SSC Server MUST cancel all subscriptions of the specific SSC Client.

Example:

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

When applications of SSC Servers need to regulate SSC Clients, for example to enforce exclusive access for a single Client at a time, this Method SHOULD be used to detect disconnecting Clients even when the transport layer works connection-less. The SSC Server would see the first SSC Method Call of each unknown SSC Client as an implicit request to establish a logical connection, which the Server may accept or refuse by sending an error response, and then close the connection when this Method is called (example application: Infra-red sync of a wireless microphone).

5.1.9 SSC subscriptions - /osc/state/subscribe

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

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

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

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

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

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

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

5.1.9.1 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 notifi-

cations 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.

5.1.9.2 Subscription cancelling and expiration

The SSC Server MUST terminate a subscription in these cases:

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

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

Optional subscription request parameters related to termination:

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

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

5.1.9.3 Subscribing to multiple addresses

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

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

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

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

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

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

Supscription request and reply syntax

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

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

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

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

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

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

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

5.1.9.4 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, "timeout": 3600 },
          "out1": { "xlr2": { "level": null }}
    } ] }}}
RX: { "osc": { "state": { "subscribe": [ {
          "#": { "min": 100, "max": 50, "timeout": 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 }}},
                      { }
                ]}
          ] } } ]
    } }
```

5.1.10 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" }}
```

5.1.11 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.

5.1.12 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 4.3.5.

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

... 3 seconds pass ..

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

5.1.13 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.

5.1.14 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"}] }}] }}
```

5.1.15 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.

5.1.15.1 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.

5.1.15.2 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 }}}
```

5.1.15.3 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 }}}
```

5.1.15.4 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": { "subscription": null }}}
RX: { "osc": { "feature": { "subscription": false }}}
```

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; if it has a menu, this name should be configurable.

If the device is networked, this name shall be used as the name for device discovery. 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.

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`.

5.2.10.3.1 /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.

5.2.10.3.2 /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.

5.2.10.3.3 /device/network/ipv4/ipaddr

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

5.2.10.3.4 /device/network/ipv4/netmask

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

5.2.10.3.5 /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.

5.2.10.4 /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.

5.2.10.4.1 /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.

5.2.10.4.2 /device/network/ipv6/ipaddr

Read-only array of all the IPv6 addresses used on the specified physical interfaces in standard string notation.

6. SSC Transport Layer Adaptations

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

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

6.1 UDP/IP

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

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

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

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

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

6.2 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 HTTP(S)/TCP/IP

Support for transporting SSC Messages over HTTP over TCP is OPTIONAL. If an SSC Server provides the HTTP transport, it MUST also provide TCP transport. Consequentially, HTTP MUST be provided over the same set of IP versions as UDP. An SSC Server supporting HTTP SHOULD support HTTP version 1.1.

HTTP should be provided by those SSC Server devices that should be easily accessible for control apps based on smart phone, tablets, or web-apps (might be served by the device itself).

SSC Messages are transported as HTTP request or reply contents. The MIME-Type of the contents MUST be specified as "application/json" in the HTTP header field "Content-Type". Using this MIME-Type eases integration of SSC-over-HTTP into Web-Apps using JSON libraries.

The SSC Client MUST send SSC Messages to the SSC Server as the contents of HTTP POST requests. The Reply of the server is transported back as the contents of the HTTP response.

There are two different models of SSC communication over HTTP:

One SSC Message at a time:

The SSC Client formats the HTTP request including SSC Message as contents, and specifies the length of the SSC Message in the HTTP "Content-Length"-header. The SSC Server sends the SSC Reply Message in the HTTP response, also specifying the "Content-Length" header. The HTTP connection may persist, depending on the HTTP "Connection"-header.

Every SSC Client and SSC Server supporting the HTTP transport MUST implement this.

SSC Message stream over HTTP:

Here the client doesn't specify the "Content-Length", but instead uses HTTP "chunked" Transport Encoding. In this case, one HTTP chunk MUST contain one SSC Message, and the client can send multiple successive SSC Messages as the contents of a single HTTP request. The SSC Server SHALL use chunked transport encoding for the HTTP response as well, also using one HTTP transport chunk for one SSC Message. The SSC Client can terminate the SSC Message stream by closing the HTTP request with a zero-size chunk. At this moment, the Server SHALL also close the ongoing HTTP response. The HTTP connection MAY still persist, depending on the HTTP "Connection" header.

SSC Clients and SSC Servers SHOULD implement this transport mechanism, if a high rate of SSC Message transactions is expected, because it offers significantly less protocol overhead. The SSC Server MUST refuse the HTTP request for chunked transport encoding if it does not support the SSC Message transport described here. The SSC Client MUST support chunked HTTP responses if it requests chunked transport from the SSC Server.

The HTTP request URL to be used by the SSC Client is expected to be discovered by the SSC Client by means of the server discovery protocol. The default URL is "/ssc".

The address spaces of SSC and HTTP are treated as interlinked: leading parts of SSC addresses may be appended to the HTTP request URL, meaning that this part of the SSC addresses is left out of the SSC Messages transported over this specific HTTP request. This means that this HTTP transaction:

```
TX: POST /ssc/out1/xlr2 HTTP/1.1
    Host: ssc-server.local.
    Content-Type: application/json

    { "gain": -10 }
RX: HTTP/1.1 OK
    Content-Type: application/json

    { "gain": -10 }
```

is equivalent to this:

```
TX: POST /ssc HTTP/1.1
    Host: ssc-server.local.
    Content-Type: application/json

    { "out1": { "xlr2": { "gain": -10 }}}
RX: HTTP/1.1 OK
    Content-Type: application/json

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

SSC errors and HTTP errors are strictly separate.

The TCP port number to be used by the SSC Client is expected to be discovered by the client by means of the server discovery protocol. The default port number is 80, same as for standard HTTP, or 443 for HTTPS.

Rationale: The standard HTTP ports are least likely to be blocked by firewall setups. The SSC service can easily coexist with other HTTP services on the same device by utilising a separated HTTP URI base, so a separate port should not be needed.

HTTP may be provided by the SSC Server as HTTP-over-TLS, optionally secured with server or even client side certificates, if absolute security is required for the SSC system.

6.4 Secure Shell Transport/TCP/IP

This is handled exactly like TCP/IP transport.

6.5 SSC Server Discovery

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

The DNS Service-Type is specified as "_ssc".

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

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

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

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

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

The following properties are currently defined for the TXT record:

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

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

6.6 IEEE 802.15.4 / ZigBee / DECT

One SSC Message per transport packet.

6.7 Low-bandwidth serial infrared link

Tbd. Framing defined separately.

6.8 Byte-stream connections (serial interface etc.)

SSC Messages are separated from the byte stream in the same way as used for TCP/IP connections. Physical parameters of the serial link are out of the scope of this specification.

6.9 Unidirectional low-bandwidth monitoring

Tbd. Device has defined set of addresses which are permanently monitored.

6.10 Configuration files

Configuration files should be kept readable and editable for humans.

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

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

Normally, a configuration file would contain only one SSC Message, encompassing all configurable settings. Multiple SSC Messages have the same semantics as if they were sent to an SSC Server in the sequence as they appear in the file. Allowing multiple Messages in configurations is useful for log-structured config files, where each setting may be appended, one at a time, as they are configured by the user handling the device.

Further syntactic elements are allowed in SSC files:

"#" as the first character on a line introduces a comments line, which should be ignored by the SSC parser.

As a special comment, "#!" on the first line of a file is interpreted in the UNIX way to indicate the application that should handle the file, so that the files may be used as executables.

6.11 Scripting files

See configuration files. Normally, multiple SSC Messages wil be contained in a script, one after the other. Allow meta-commands for script player:

```
#! osc: { delay: 3 }
```

Time stamps may be inserted by SSC logger:

```
#! osc: { timestamp: "2012-07-26T12:51:22+0200" }
```

To play back recorded log with logged pauses, starttime-meta-command allows SSC script player to re-interpret logged time stamps as relative delays:

```
#! osc: { starttime: "2012-07-26T12:35:46+02:00" }
```

7. Appendix

7.1 Relaxed SSC Parser for Interactive Use

An SSC Server may implement the following relaxed parsing rules to interpret interactive input.

SSC Messages generated by the SSC Server or SSC Messages generated by programs must be strictly conformant to this specification.

- the top-level braces of an SSC message may be left out.
- the double quotes surrounding SSC address parts may be omitted.
- the JSON value null that indicates SSC queries may be left out.
- SSC address parts may be concatenated with "/", and inner JSON objects may be left out, when only a single SSC Method is addressed in an SSC Message.

The relaxed rules allow these SSC transactions:

```
TX: { "osc": { "state": { "base": "/out1/xlr2" }}}
RX: { "osc": { "state": { "base": "/out1/xlr2" }}}
TX: { "gain": -10 }
RX: { "gain": -10 }
TX: { "mute": null }
RX: { "mute": false }
```

to be entered interactively in this even more intuitive way:

```
TX: osc/state/baseaddr: "/out1/xlr2"
RX: { "osc": { "state": { "baseaddr": "/out1/xlr2" }}}
TX: gain: -10
RX: { "gain": -10 }
TX: mute:
RX: { "mute": false }
```

7.2 References

7.2.1 Normative References

tbd.

7.2.2 Additional References

tbd.

8. Developer‘s Guide for evolution wireless D1

This document describes in detail how a developer should use the SSC interface as implemented for evolution wireless D1.

9. Limitations

9.1 SSC Transport Layer

The SSC Server implemented for evolution wireless D1 devices supports only UDP/IP as transport protocol. All the devices support both IPv4 and IPv6.

9.2 Subscriptions

Evolution wireless D1 receivers support SSC Method subscriptions from up to eight different SSC clients simultaneously.

10. SSC Method List

10.1 /interface/version

Parameter type: String
Permission: Read only

Pre-condition: N.A.
Post-condition: N.A.

Description: SSC interface version.

Example:

```
TX: {"interface":{"version":null}}
RX: {"interface":{"version":"1.0"}}
```

10.2 /device/identity/product

Parameter type: String
Permission: Read only

Pre-condition: N.A.
Post-condition: N.A.

Description: Product identification, should be identical to the designation on the label on the product itself or to the included electronic/component.

Example:

```
TX: {"device":{"identity":{"product":null}}}
RX: {"device":{"identity":{"product":"EWD1"}}}
```

10.3 /device/identity/version

Parameter type: String
Permission: Read only

Pre-condition: N.A.
Post-condition: N.A.

Description: Device firmware version.

Example:

```
TX: {"device":{"identity":{"version":null}}}
RX: {"device":{"identity":{"version":"2.0.0"}}}
```

10.4 /device/identity/serial

Parameter type: String
Permission: Read only

Pre-condition: N.A.
Post-condition: N.A.

Description: Unique product number, identical to the number on a product label. The value is given by production.

Example:

```
TX: {"device":{"identity":{"serial":null}}}
RX: {"device":{"identity":{"serial":"1454100930"}}}
```

10.5 /device/identity/vendor

Parameter type: String
Permission: Read only

Pre-condition: N.A.
Post-condition: N.A.

Description: Device vendor name.

Example:

```
TX: {"device":{"identity":{"vendor":null}}}
RX: {"device":{"identity":{"vendor":"Sennheiser electronic GmbH & Co. KG"}}}
```

10.6 /device/name

Parameter type: String
Permission: Read/Write, Subscribe-able

Pre-condition: N.A.
Post-condition: N.A.

Description: 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; if it has a menu, this name should be configurable.

If the device is networked, this name shall be used as the name for device discovery. 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.

Example:

```
TX: {"device":{"name":null}}
RX: {"device":{"name":"ewD1"}}
```

10.7 /device/language

Parameter type: String
Permission: Read/Write

Pre-condition: N.A.
Post-condition: N.A.

Description: 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. 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.

Example:

```
TX: {"device":{"language":null}}
RX: {"device":{"language":["en_GB"]}}
```

10.8 /device/network/ether/interfaces

Parameter type: String
Permission: Read only

Pre-condition: N.A.
Post-condition: N.A.

Description: Read-only array containing a list of all the user-relevant Ethernet interfaces of the device.

Example:

```
TX: {"device":{"network":{"ether":{"interfaces":null}}}}
RX: {"device":{"network":{"ether":{"interfaces":["LAN"]}}}}
```

10.9 /device/network/ether/macs

Parameter type: String
Permission: Read only

Pre-condition: N.A.
Post-condition: N.A.

Description: 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.

Example:

```
TX: {"device":{"network":{"ether":{"macs":null}}}}
RX: {"device":{"network":{"ether":{"macs":["00:1B:66:7D:F8:99"]}}}}
```

10.10 /device/network/ipv4/interfaces

Parameter type: Number
Permission: Read only

Pre-condition: N.A.
Post-condition: N.A.

Description: Read-only array relating to /device/network/ether/interfaces. The array contains numbers indexing into the array of physical interface names. The interface index array contains the indices of all physical interfaces which share the IPv4 configuration.

Example:

```
TX: {"device":{"network":{"ipv4":{"interfaces":null}}}}
RX: {"device":{"network":{"ipv4":{"interfaces":[1]}}}}
```

10.11 /device/network/ipv4/auto

Parameter type: Boolean
Permission: Read/Write

Pre-condition: N.A.
Post-condition: N.A.

Description: Array containing a list of boolean values that indicates whether IPv4 shall be configured automatically by means of DHCP and ZeroConf (Auto-IP) of all the user-relevant Ethernet interface of the device. The order of the list matches /device/network/ether/interfaces. If the value is set to false the values of /device/network/ipv4/fixed_ipaddr, /device/network/ipv4/fixed_ipaddr and /device/network/ipv4/fixed_ipaddr will be used during target initialization at start-up. A value change takes effect after device reset!

Example:

```
TX: {"device":{"network":{"ipv4":{"auto":null}}}}
RX: {"device":{"network":{"ipv4":{"auto":[true]}}}}
```

10.12 /device/network/ipv4/ipaddr

Parameter type: String
Permission: Read only

Pre-condition: N.A.
Post-condition: N.A.

Description: Read-only array containing a list of the current IPv4 addresses of all the user-relevant Ethernet interface of the device. The order of the list matches /device/network/ether/interfaces. The IPv4 addresses are specified as strings in standard dot-decimal notation.

Example:

```
TX: {"device":{"network":{"ipv4":{"ipaddr":null}}}}
RX: {"device":{"network":{"ipv4":{"ipaddr":["192.168.1.203"]}}}}
```

10.13 /device/network/ipv4/netmask

Parameter type: String
Permission: Read only

Pre-condition: N.A.
Post-condition: N.A.

Description: Read-only array containing a list of the current IPv4 netmasks of all the user-relevant Ethernet interface of the device. The order of the list matches /device/network/ether/interfaces. The IPv4 netmasks are specified as strings in standard dot-decimal notation.

Example:

```
TX: {"device":{"network":{"ipv4":{"netmask":null}}}}
RX: {"device":{"network":{"ipv4":{"netmask":["255.255.255.0"]}}}}
```

10.14 /device/network/ipv4/gateway

Parameter type: String
Permission: Read only

Pre-condition: N.A.
Post-condition: N.A.

Description: Read-only array containing a list of the IPv4 gateways of all the user-relevant Ethernet interface of the device. The order of the list matches /device/network/ether/interfaces. The IPv4 gateways are specified as strings in standard dot-decimal notation.

Example:

```
TX: {"device":{"network":{"ipv4":{"gateway":null}}}}
RX: {"device":{"network":{"ipv4":{"gateway":["192.168.1.1"]}}}}
```

10.15 /device/network/ipv4/fixed_ipaddr

Parameter type: String
Permission: Read/Write, Subscribe-able

Pre-condition: N.A.
Post-condition: N.A.

Description: An array containing a list of the stored IPv4 addresses in EEPROM of all the user-relevant Ethernet interface of the device. The order of the list matches /device/network/ether/interfaces. The IPv4 addresses are specified as strings in standard dot-decimal notation. The stored IPv4 addresses in EEPROM are used by the device if /device/network/ipv4/auto is set to false during start-up of the device.

Example:

```
TX: {"device":{"network":{"ipv4":{"fixed_ipaddr":["192.168.1.203"]}}}}
RX: {"device":{"network":{"ipv4":{"fixed_ipaddr":["192.168.1.203"]}}}}
```

10.16 /device/network/ipv4/fixed_netmask

Parameter type: String
Permission: Read/Write, Subscribe-able

Pre-condition: N.A.
Post-condition: N.A.

Description: An array containing a list of the stored IPv4 netmasks in EEPROM of all the user-relevant Ethernet interface of the device. The order of the list matches /device/network/ether/interfaces. The IPv4 addresses are specified as strings in standard dot-decimal notation. The stored IPv4 netmasks in EEPROM are used by the device if /device/network/ipv4/auto is set to false during start-up of the device.

Example:

```
TX: {"device":{"network":{"ipv4":{"fixed_netmask":["255.255.255.0"]}}}}
RX: {"device":{"network":{"ipv4":{"fixed_netmask":["255.255.255.0"]}}}}
```

10.17 /device/network/ipv4/fixed_gateway

Parameter type: String
Permission: Read/Write, Subscribe-able

Pre-condition: N.A.
Post-condition: N.A.

Description: An array containing a list of the stored IPv4 gateways in EEPROM of all the user-relevant Ethernet interface of the device. The order of the list matches /device/network/ether/interfaces. The IPv4 addresses are specified as strings in standard dot-decimal notation. The stored IPv4 gateways in EEPROM are used by the device if /device/network/ipv4/auto is set to false during start-up of the device.

Example:

```
TX: {"device":{"network":{"ipv4":{"fixed_gateway":["192.168.1.1"]}}}}
RX: {"device":{"network":{"ipv4":{"fixed_gateway":["192.168.1.1"]}}}}
```

10.18 /device/network/ipv6/interfaces

Parameter type: Number
Permission: Read only

Pre-condition: N.A.
Post-condition: N.A.

Description: Read-only array relating to /device/network/ether/interfaces. The array contains numbers indexing into the array of physical interface names. The interface index array contains the indices of all physical interfaces which share the IPv6 configuration.

Example:

```
TX: {"device":{"network":{"ipv6":{"interfaces":null}}}}
RX: {"device":{"network":{"ipv6":{"interfaces":[1]}}}}
```

10.19 /device/network/ipv6/ipaddr

Parameter type: String
Permission: Read only

Pre-condition: N.A.
Post-condition: N.A.

Description: Read-only array containing an arrary of all the IPv6 addresses of all the user-relevant Ethernet interface of the device. The order of the list matches /device/network/ether/interfaces. The IPv6 addresses are specified as strings in standard notation. Each IPv6 network interface can contain maximal 3 IPv6 addresses.

Example:

```
TX: {"device":{"network":{"ipv6":{"ipaddr":null}}}}
RX: {"device":{"network":{"ipv6":{"ipaddr":[["fe80::21b:66ff:fe7d:f899",
     "2001:db8::b2f4:4bff:fa71:1a56"]]}}}}
```

10.20 /device/reset

Parameter type: Boolean
Permission: Read/Write

Pre-condition: N.A.
Post-condition: N.A.

Description: A soft reset may be necessary for some configuration settings to take effect.

Example:

```
TX: {"device":{"reset":true}}
RX: {"device":{"reset":true}}
```

10.21 /device/factory_reset

Parameter type: Boolean
Permission: Read/Write

Pre-condition: N.A.
Post-condition: The device will restart itself so that after restart the new settings are used.

Description: Boolean value to re-configure the device with factory defaults.

Example:

```
TX: {"device":{"factory_reset":true}}
RX: {"device":{"factory_reset":true}}
```

10.22 /osc/error

Parameter type: Number (limit range)
Permission: Read only

Pre-condition: N.A.
Post-condition: N.A.

Description: 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.

The error message MUST contain an integer numeric value, the error code. The error code SHOULD be chosen from the SSC Error List detailed in chapter 2.

Examples:

```
TX: {"out1":{"gain":10}}
RX: {"osc":{"error":[{"out1":[404]}]}}     <-- not found

TX: {"write_protection":true}
RX: {"osc":{"error":[{"write_protection":[406]}]}}    <-- read only

TX: {"osc":{"limits":[{"internal":{"debug_screen":null}}]}}
RX: {"osc":{"error":[{"osc":{"limits":[454]}}]}}    <-- hidden
```

10.23 /osc/xid

Parameter type: N.A.
Permission: N.A.

Pre-condition: N.A.
Post-condition: N.A.

Description: When an SSC Client 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.

Examples:

```
TX:{"osc":{"xid":1234567890,"ping":["AbCdEfGhIjKlMnOpQrStUvWxYz",3,14159265358979323846
26433832795]}}
RX:  {"osc":{"ping":["AbCdEfGhIjKlMnOpQrStUvWxYz",3,14159265358979323846264338327 95],
"xid":1234567890}}

TX: {"osc":{"xid":1234567890},"brightness":null}
RX: {"brightness":75,"osc":{"xid":1234567890}}
```

10.24 /osc/version

Parameter type: String.
Permission: Read only.

Pre-condition: N.A.
Post-condition: N.A.

Description: Reports the SSC version implemented in the server.

Example:

```
TX: {"osc":{"version":null}}
RX: {"osc":{"version":"1.0"}}
```

10.25 /osc/feature/pattern

Parameter type: Boolean.
Permission: Read only.

Pre-condition: N.A.
Post-condition: N.A.

Description: 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

Example:

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

10.26 /osc/schema

Parameter type: N.A.
Permission: N.A..

Pre-condition: N.A.
Post-condition: N.A.

Description: The /osc/schema method exists to allow clients to query servers about what address schemes are available on a specific server. SSC clients MUST be able to understand both bundled and unbundled replies. 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.

Examples:

```
TX: {"osc":{"schema":null}}
RX:   {"osc":{"schema":[{"audio":{},"device":{},"mates":{},"rx1":{},"osc":{},"bright-
ness":null}]}}
TX: {"osc":{"schema":[{"rx1":null}]}}
RX:{"osc":{"schema":[{"rx1":{"warnings":null,"walktest":null,"rf_quality":null,"pair":null,
"mute_switch_active":null,"identify"
   :null,"autolock":null}}]}}
```

10.27 /osc/feature/subscription

Parameter type: Boolean.
Permission: Read only.

Pre-condition: N.A.
Post-condition: N.A.

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

Example:

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

10.28 /osc/limits

Parameter type: N.A.
Permission: N.A..

Pre-condition: N.A.
Post-condition: N.A.

Description: 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.

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

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
- units string String describing value units (preferably SI)
- desc string descriptive text, meant for display to the user
- 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.

Examples:

```
TX: {"osc":{"limits":[{"brightness":null}]}}
RX:  {"osc":{"limits":[{"brightness":[{"type":"Number","max":100,"min":0,"inc":1,"uni
ts":"%"}]}]}}

TX: {"osc":{"limits":[{"audio":{"equalizer":{"preset":null}}}]}}
RX:    {"osc":{"limits":[{"audio":{"equalizer":{"preset":[{"type":"Number","desc":"EQ
presets","option":[0,1,2,3,4,5,6,7,8,9,10,11,12,13],"option_desc":["Off","Cus-
tom","Vocals","Presence Boost","Mid Cut 1","Mid Cut 2","High Mid Cut","Low Mid
Cut","High Boost","High Cut","Megaphone","Telephone","Acoustic Guitar 1","Acoustic
Guitar 2"]}]}}}]}}
```

10.29 /osc/feature/baseaddr

Parameter type: Boolean.
Permission: Read only.

Pre-condition: N.A.
Post-condition: N.A.

Description: Using a baseaddr helps to explore a device in a truly interactive manner, and may additionally be used to reduce message lengths by shortening addresses in SSC requests.

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

Example:

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

10.30 /osc/state/close

Parameter type: Boolean.
Permission: Write only.

Pre-condition: N.A.
Post-condition: N.A.

Description: When an SSC Client calls this SSC Method with a true argument, the SSC Server MUST close the connection immediately after the reply has been sent.

Example:

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

10.31 /osc/state/prettyprint

Parameter type: Boolean.
Permission: Read only.

Pre-condition: N.A.
Post-condition: N.A.

Description: 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

Examples:

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

Example

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

10.32 /osc/feature/timetag

Parameter type: Boolean.
Permission: Read only.

Pre-condition: N.A.
Post-condition: N.A.

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

Example:

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

10.33 /osc/state/subscribe

Parameter type: N.A.
Permission: N.A.

Pre-condition: N.A.
Post-condition: N.A.

Description: 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.

Examples:

```
TX: {"osc":{"xid":1234567890,"state":{"subscribe":[{"brightness":null}]}}}
RX: {"osc":{"xid":1234567890},"osc":{"state":{"subscribe":[{"brightness":null}]}}}
```

Subscribing with non-default parameters:

```
TX: {"osc":{"state":{"subscribe":[{"#":{"lifetime":2000},"brightness":null}]}}}
RX: {"osc":{"state":{"subscribe":[{"#":{"lifetime":2000},"brightness":null}]}}}
RX: {"brightness":75}
```

Parameter value changes:

```
RX: {"brightness":70}
RX: {"brightness":75}
RX: {"brightness":70}
RX: {"brightness":65}
```

subscription terminates:

```
RX: {"osc":{"error":[{"brightness":[310]}]}}
TX: {"osc":{"state":{"subscribe":[{"#":{"lifetime":2000},"rx1":{"mute_switch_active":
null},"mates":{"tx1":{"bat_lifetime"
    :null,"bat_gauge":null,"switch1":{"state":null}}}}]}}}
RX: {"osc":{"state":{"subscribe":[{"#":{"lifetime":2000},"rx1":{"mute_switch_active":
null},"mates":{"tx1":{"bat_gauge"
    :null}},"mates":{"tx1":{"bat_lifetime":null}},"mates":{"tx1":{"switch1":{"state":
null}}}}]}}}'
RX:{"rx1":{"mute_switch_active":false},"mates":{"tx1":{"bat_gauge":83}},"mates":{"tx-
1":{"bat_lifetime"
    :38460}},"mates":{"tx1":{"switch1":{"state":true}}}}
```

10.34 /device/max_rf_power_level

Parameter type: Number (limit range)
Permission: Read only, Subscribe-able

Pre-condition: N.A.
Post-condition: N.A.

Description: Method to retrieve the maximum RF power level of the sRX1 in mW. Supported maximum RF power levels are 1mW, 10mW and 100mW.

Example:

```
TX: {"device":{"max_rf_power_level":null}}
RX: {"device":{"max_rf_power_level":10}}
```

10.35 /brightness

Parameter type: Number (limit range)
Permission: Read/Write, Subscribe-able

Pre-condition: The parameter value must an integer value in the range [0..100]
Post-condition: The change has immediately effect.

Description: Method to retrieve/modify the OLED display brightness in percentage of the sRX1. This parameter is related to the sRX1 UI menu item "Brightness" under "System Settings".

Example:

```
TX: {"brightness":100}
RX: {"brightness":100}
```

10.36 /mates/active

Parameter type: Boolean
Permission: Read only, Subscribe-able

Pre-condition: N.A.
Post-condition: N.A.

Description: Method to get active mates. Returns ["tx1"] if the sRX1 has an established link with a transmitter, else it returns [].

Example:

```
TX: {"mates":{"active":null}}
RX: {"mates":{"active":["tx1"]}}
```

10.37 /mates/tx1/bat_state

Parameter type: Number
Permission: Read only, Subscribe-able

Pre-condition: Link must be established between transmitter and sRX1.
Post-condition: N.A.

Description: Method to retrieve status of the battery. If the connected transmitter uses a rechargeable battery and the lifetime is available the SSC Server will return with /mates/tx1/bat_lifetime, in all other cases it will return with /mates/tx1/bat_gauge. Note that it takes some time before it is possible to retrieve the predicted lifetime of a rechargeable battery, before the lifetime is available the actual capacity of the rechargeable battery is used.

Examples:

```
TX: {"mates":{"tx1":{"bat_state":null}}}
RX: {"mates":{"tx1":{"bat_gauge":16}}}

TX: {"mates":{"tx1":{"bat_state":null}}}
RX: {"mates":{"tx1":{"bat_lifetime":13560}}}
```

10.38 /mates/tx1/bat_type

Parameter type: Number (limit range)
Permission: Read only, Subscribe-able

Pre-condition: Link must be established between transmitter and sRX1.
Post-condition: N.A.

Description: Method to retrieve type of the battery.

Index	Description
0	Battery
1	Rechargeable

Examples:

```
TX: {"mates":{"tx1":{"bat_type":null}}}
RX: {"mates":{"tx1":{"bat_ type":0}}}
```

10.39 /mates/tx1/bat_gauge

Parameter type: Number
Permission: Read only, Subscribe-able

Pre-condition: Link must be established between transmitter and sRX1.
Post-condition: N.A.

Description: Method to retrieve the capacity of the rechargeable battery, or in case a battery is used the battery voltage level in percentage.

Examples:

```
TX: {"mates":{"tx1":{"bat_gauge":null}}}
RX: {"mates":{"tx1":{"bat_gauge":100}}}
```

10.40 /mates/tx1/bat_lifetime

Parameter type: Number
Permission: Read only, Subscribe-able

Pre-condition: Link must be established between transmitter, powered by a rechargeable battery, and sRX1.
Post-condition: N.A.

Description: Method to retrieve the lifetime of the rechargeable battery in seconds.

Examples:

```
TX: {"mates":{"tx1":{"bat_lifetime":null}}}
RX: {"mates":{"tx1":{"bat_lifetime":12900}}}
```

10.41 /mates/tx1/acoustic

Parameter type: String
Permission: Read only, Subscribe-able

Pre-condition: Link must be established between transmitter and sRX1.
Post-condition: N.A.

Description: Method to get the acoustic input type of mate "tx1". Support transmitters are: "Mic", "Line", "MME865", "MD42", "MMD945", "MMD935", "MMD845", "MMD835", "MMD815_1", "MMK965s", "MMK965c". When a not supported capsule is connected "INCOMPATIBLE" will be returned, and if there is no capsule connected "" will be returned.

Examples:

Not connected with a transmitter:

```
TX: {"mates":{"tx1":{"acoustic":null}}}
RX: {"mates":{"tx1":{"acoustic":""}}}
```

Connected with a MD42:

```
TX: {"mates":{"tx1":{"acoustic":null}}}
RX: {"mates":{"tx1":{"acoustic":"MD42"}}}
```

10.42 /mates/tx1/switch1/label

Parameter type: String
Permission: Read only

Pre-condition: Link must be established between transmitter and sRX1.
ost-condition: N.A.

Description: Method to get the switch1 label on the transmitter.

Example:

```
TX: {"mates":{"tx1":{"switch1":{"label":null}}}}
RX: {"mates":{"tx1":{"switch1":{"label":"Mute"}}}}
```

10.43 /mates/tx1/warnings

Parameter type: String
Permission: Read only, Subscribe-able

Pre-condition: Link must be established between transmitter and sRX1.
Post-condition: N.A.

Description: Method to get transmitter warnings connected to the sRX1. Supported warnings are "" if there is no warning or "Low Bat" if the capacity of a rechargeable battery is below a certain level or if the voltage of a battery is below a certain level, the levels are depending on the battery manufacturer and type of transmitter.

Example:

```
TX: {"mates":{"tx1":{"warnings":null}}}
RX: {"mates":{"tx1":{"warnings":["Low Bat"]}}}
```

10.44 /mates/tx1/switch1/state

Parameter type: Boolean
Permission: Read only, Subscribe-able

Pre-condition: Link must be established between transmitter and sRX1.
Post-condition: N.A.

Description: Method to get the switch1 state (Mute switch) on the transmitter. Note that the sRX1 can have disabled switch1 with the /rx1/mute_switch_active method.

Example:

```
TX: {"mates":{"tx1":{"switch1":{"state":null}}}}
RX: {"mates":{"tx1":{"switch1":{"state":true}}}}
```

10.45 /mates/tx1/device_type

Parameter type: Number (limit range)
Permission: Read, Subscribe-able

Pre-condition: /mates/tx1 must be active.
Post-condition: NA.

Description: Method to retrieve the transmitter type.

Index	Description
0	Handheld
1	Bodypack
2	Tablestand
3	Boundary

Example:

```
TX: {"mates":{"tx1":{"device_type":null}}}
RX: {"mates":{"tx1":{"device_type":2}}}
```

10.46 /rx1/autolock

Parameter type: Boolean
Permission: Read/Write, Subscribe-able

Pre-condition: N.A.
Post-condition: The change has immediately effect.

Description: Method to modify/get the auto-lock state of the sRX1. This parameter is related to the sRX1 UI menu item "Auto Lock" under "System Settings".

Example:

```
TX: {"rx1":{"autolock":null}}
RX: {"rx1":{"autolock":true}}
```

10.47 /rx1/warnings

Parameter type: String
Permission: Read only, Subscribe-able

Pre-condition: Link must be established between transmitter and sRX1.
Post-condition: N.A.

Description: Method to get sRX1 warnings. Supported warnings are "" if there is no warning, "HW Failure" in case during device start-up a HW failure is detected, "Bad Link" if the connected between the transmitter and sRX1 is bad, "No Link" if there is no transmitter connected with the sRX1 or "No FW Image" if the sRX1 cannot find a firmware image to update the connected transmitter via FWU_OTA.

Example:

```
TX: {"rx1":{"warnings":null}}
RX: {"rx1":{"warnings":["Bad Link"]}}
```

10.48 /rx1/pair

Parameter type: Boolean
Permission: Read/Write, Subscribe-able

Pre-condition: The sRX1 should not be in walk-test mode.
Post-condition: The change has immediately effect.

Description: Method to start/stop the pair mode on the sRX1.

Example:

```
TX: {"rx1":{"pair":true}}
RX: {"rx1":{"pair":true}}
```

10.49 /rx1/identify

Parameter type: Boolean
Permission: Read/Write, Subscribe-able

Pre-condition: The sRX1 should not be in pairing or walk-test mode.
Post-condition: The change has immediately effect.

Description: Method to start/stop the identify mode on the sRX1.

Example:

```
TX: {"rx1":{"identify":true}}
RX: {"rx1":{"identify":true}}
```

10.50 /rx1/walktest

Parameter type: Boolean
Permission: Read/Write, Subscribe-able

Pre-condition: The sRX1 should not be in pairing or identify mode.
Post-condition: The change has immediately effect.

Description: Method to start/stop the walk-test mode on the sRX1.

Example:

```
TX: {"rx1":{"walktest":true}}
RX: {"rx1":{"walktest":true}}
```

10.51 /rx1/rf_quality

Parameter type: Number
Permission: Read only, Subscribe-able

Pre-condition: N.A.
Post-condition: N.A.

Description: Method to the RF quality in percentage.

Example:

```
TX: {"rx1":{"rf_quality":null}}
RX: {"rx1":{"rf_quality":50}}
```

10.52 /rx1/mute_switch_active

Parameter type: Boolean
Permission: Read/Write, Subscribe-able

Pre-condition: N.A.
Post-condition: The change has immediately effect.

Description: Method to enable/disable the mute switch functionality on the transmitter, if a mute switch is available. This parameter is related to the sRX1 UI menu item "Mute Switch" under "System Settings".

Example:

```
TX: {"rx1":{"mute_switch_active":false}}
RX: {"rx1":{"mute_switch_active":false}}
```

10.53 /rx1/rf_stack_active

Parameter type: Boolean
Permission: Read/Write, Subscribe-able

Pre-condition: N.A.
Post-condition: The change has immediately effect.

Description: Method to enable/disable the RF stack on the receiver.

Example:

```
TX: {"rx1":{"rf_stack_active":null}}
RX: {"rx1":{"rf_stack_active ":true}}
```

10.54 /audio/out1/label

Parameter type: String
Permission: Read only

Pre-condition: N.A.
Post-condition: N.A.

Description: Method to retrieve the output label.

Example:

```
TX: {"audio":{"out1":{"label":null}}}
RX: {"audio":{"out1":{"label": ["AF OUT BAL +18 dBu max","AF OUT UNBAL"]}}}
```

10.55 /audio/out1/level_db

Parameter type: Number (limit range)
Permission: Read only, Subscribe-able

Pre-condition: N.A.
Post-condition: N.A.

Description: Method to retrieve the actual output level on the sRX1. The parameter value is inside a defined range [-80..0].

Example:

```
TX: {"audio":{"out1":{"level_db":null}}}
RX: {"audio":{"out1":{"level_db":-56}}}
```

10.56 /audio/low_cut

Parameter type: Boolean
Permission: Read/Write, Subscribe-able

Pre-condition: N.A.
Post-condition: The change has immediately effect.

Description: Method to enable/disable low cut settings, equivalent to the "Low Cut" menu item under "Audio Settings" in the sRX1 UI.

Example:

```
TX: {"audio":{"low_cut":null}}
RX: {"audio":{"low_cut":false}}
```

10.57 /audio/equalizer/custom

Parameter type: Number
Permission: Read/Write, Subscribe-able

Pre-condition: N.A.
Post-condition: The change has immediately effect.

Description: Custom EQ preset gains. An array with 7 integer values between -12 and 12 (dB) must be provided to set the gains. If the equalizer custom preset is not selected, it will be selected automatically.

Example:

```
TX: {"audio":{"equalizer":{"custom":[0,-10,-8,12,0,0,0]}}}
RX: {"audio":{"equalizer":{"custom":[0,-10,-8,12,0,0,0]}}}
```

10.58 /audio/effects_reset

Parameter type: Boolean
Permission: Read/Write

Pre-condition: N.A.
Post-condition: The change has immediately effect.

Description: Method to restore the audio effects default settings, equivalent to the "Effects Reset" menu item under "Audio Settings" in the sRX1 UI.

Example:

```
TX: {"audio":{"effects_reset":true}}
RX: {"audio":{"effects_reset":true}}
```

10.59 /device/state

Parameter type: Number (limit range)
Permission: Read only, Subscribe-able

Pre-condition: N.A.
Post-condition: N.A.

Description: Method to retrieve the state of the sRX1.

Index	Description
0	Normal
1	Pairing
2	Receiver Update
3	Transmitter Update
4	Transmitter Update Confirmation

Examples:

```
TX: {"device":{"state":null}}
RX: {"device":{"state":4}}
```

10.60 /device/progress

Parameter type: Number (limit range)
Permission: Read only, Subscribe-able

Pre-condition: The sRX1 must be in pairing or (receiver/transmitter) update state.
Post-condition: N.A.

Description: Method to retrieve the displayed progress bar in the sRX1 UI. The parameter value is a defined range [0..100].

Example:

```
TX: {"device":{"progress":null}}
RX: {"device":{"progress":4}}
```

10.61 /device/update/confirmation

Parameter type: Boolean
Permission: Read/Write

Pre-condition: The sRX1 must be in transmitter update confirmation state.
Post-condition: The change has immediately effect.

Description: Method to approve or reject a transmitter update.

Example:

```
TX: {"device":{"update":{"confirmation":true}}}
RX: {"device":{"update":{"confirmation":true}}}
```

10.62 /audio/equalizer/preset

Parameter type: Number (limit range)
Permission: Read/Write, Subscribe-able

Pre-condition: The parameter value must be inside the index range [0-13].

Index	Description
0	Off
1	Custom
2	Vocals
3	Presence Boost
4	Mid Cut 1
5	Mid Cut 2
6	High Mid Cut
7	Low Mid Cut
8	High Boost
9	High Cut
10	Megaphone
11	Telephone
12	Acoustic Guitar 1
13	Acoustic Guitar 2

Post-condition: The change has immediately effect.

Description: Method to retrieve/modify the equalizer preset. This parameter is related to the sRX1 UI menu item "Equalizer" under "Audio Settings".

Example:

```
TX: {"audio":{"equalizer":{"preset":null}}}
RX: {"audio":{"equalizer":{"preset":1}}}
```

10.63 /audio/de_esser/preset

Parameter type: Number (limit range)
Permission: Read/Write, Subscribe-able

Pre-condition: The parameter value must be inside the index range [0-2].

Index	Description
0	Off
1	Broad
2	Selective

Post-condition: The change has immediately effect.

Description: Method to retrieve/modify the de-esser preset. This parameter is related to the sRX1 UI menu item "De-Esser" under "Audio Settings".

Example:

```
TX: {"audio":{"de_esser":{"preset":null}}}
RX: {"audio":{"de_esser":{"preset":0}}}
```

10.64 /audio/agc/preset

Parameter type: Number (limit range)
Permission: Read/Write, Subscribe-able

Pre-condition: The parameter value must be inside the index range [0-2].

Index	Description
0	Off
1	Hard
2	Soft

Post-condition: The change has immediately effect.

Description: Method to retrieve/modify the AGC preset. This parameter is related to the sRX1 UI menu item "Auto Gain Control" under "Audio Settings".

Example:

```
TX: {"audio":{"agc":{"preset":null}}}
RX: {"audio":{"agc":{"preset":0}}}
```

10.65 /audio/out1/gain_db

Parameter type: Number (limit range)
Permission: Read/Write, Subscribe-able

Pre-condition: The parameter value must an integer value in the range [0..30]
Post-condition: The change has immediately effect.

Description: Method to retrieve/modify the Output gain in dB. This parameter is related to the sRX1 UI menu item "Audio Level" under "Audio Settings".

Example:

```
TX: {"audio":{"out1":{"gain_db":10}}}
RX: {"audio":{"out1":{"gain_db":10}}}
```

10.66 /audio/out1/type

Parameter type: Number (limit range)
Permission: Read/Write, Subscribe-able

Pre-condition: The parameter value must be inside the index range [1,2].

Index	Description
1	Mic
2	Line

Post-condition: The change has immediately effect.

Description: Method to retrieve/modify the Output type. This parameter is related to the sRX1 UI menu item "Output Type" under "Audio Settings".

Example:

```
TX: {"audio":{"out1":{"type":null}}}
RX: {"audio":{"out1":{"type":2}}}
```

11. SSC Error List

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.

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

The error code is a three-digit integer, defined in the style of Hypertext Transfer Protocol (HTTP) response status codes, the status code is part of the HTTP/1.1 standard (RFC 7231).

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 - Successful - The action was successfully received, understood, and accepted.
- 3xx - Redirection - 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 SSC 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.

11.1 1xx Informational

The 1xx (Informational) class of status code indicates an interim response for communicating connection status or request progress prior to completing the requested action and sending a final response.

100 Continue

The client SHOULD continue with its request. This interim response is used to inform the client that the initial part of the request has been received and has not yet been rejected by the SSC Server. The client SHOULD continue by sending the remainder of the request or, if the request has already been completed, ignore this response.

The SSC Server MUST send a final response after the request has been completed.

102 Processing

The 102 (Processing) status code is an interim response used to inform the client that the SSC Server has accepted the complete request, but has not yet completed it. This status code SHOULD only be sent when the SSC Server has a reasonable expectation that the request will take significant time to complete. As guidance, if a method is taking longer than 20 seconds (a reasonable, but arbitrary value) to process the SSC Server SHOULD return a 102 (Processing) response.

The SSC Server MUST send a final response after the request has been completed.

11.2 2xx Success

This class of status code indicates that the client's request was successfully received, understood, and accepted.

200 OK

Standard response for successful requests.

201 Created

The request has been fulfilled and resulted in a new resource being created. The origin SSC Server MUST create the resource before returning the 201 status code. If the action cannot be carried out immediately, the SSC Server SHOULD respond with 202 (Accepted) response instead.

202 Accepted

The request has been accepted for processing, but the processing has not been completed. The request might or might not eventually be acted upon, as it might be disallowed when processing actually takes place. There is no facility for re-sending a status code from an asynchronous operation such as this.

210 Partial Success

The request has been partially accepted for processing.

11.3 3xx Redirection

This class of status code indicates that further action needs to be taken by the user agent in order to fulfill the request. The action required MAY be carried out by the user agent without interaction with the user.

A client SHOULD detect infinite redirection loops, since such loops generate network traffic for each redirection.

310 Subscription Terminates

The subscription is terminated on the SSC Server, because the lifetime expired or the max number of occurrences exceeded.

11.4 4xx Client Error

The 4xx class of status code is intended for cases in which the client seems to have erred.

These status codes are applicable to any request method. User agents SHOULD display any included entity to the user.

400 Bad Request

The request could not be understood by the SSC Server due to malformed syntax. The client SHOULD NOT repeat the request without modifications.

401 Unauthorized

Error code response for missing or invalid authentication token. The request requires user authentication. If the request already included Authorization credentials, then the 401 response indicates that authorization has been refused for those credentials.

403 Forbidden

Error code for user not authorized to perform the operation or the resource is unavailable for some reason (e.g. time constraints, etc.). The SSC Server understood the request, but is refusing to fulfill it. Authorization will not help and the request SHOULD NOT be repeated.

404 Not Found

The SSC Server has not found anything matching the request. No indication is given of whether the condition is temporary or permanent. The requested resource could not be found but may be available again in the future. Subsequent requests by the client are permissible. Used when the requested resource is not found, whether it doesn't exist or if there was a 401 or 403 that, for security reasons, the service wants to mask.

406 Not Acceptable (E.g. wrong type for parameter)

The SSC Server is not capable of processing the request, f.i. the client request to change a read only parameter value.

408 Request Timeout

The client did not produce a request within the time that the SSC Server was prepared to wait. The client MAY repeat the request without modifications at any later time.

409 Conflict

The request could not be completed due to a conflict with the current state of the resource. This code is only allowed in situations

where it is expected that the user might be able to resolve the conflict and resubmit the request. The response body SHOULD include enough information for the user to recognize the source of the conflict. Ideally, the response entity would include enough information for the user or user agent to fix the problem; however, that might not be possible and is not required.

410 Gone

Indicates that the resource requested is no longer available and will not be available again. This should be used when a resource has been intentionally removed and the resource should be purged. Upon receiving a 410 status code, the client should not request the resource again in the future.

413 Request Entity Too Large

The SSC Server is refusing to process a request because the request entity is larger than the SSC Server is willing or able to process.

414 Request Too Complex

The URI provided was too long for the SSC Server to process.

422 Unprocessable Entity

This status code means the SSC Server understands the content type of the request entity, and the syntax of the request entity is correct but was unable to process the contained instructions. For example, this error condition may occur if request is syntactically correct, but it is semantically erroneous.

423 Locked

The resource that is being accessed is locked.

424 Failed Dependency

The request failed due to failure of a previous request.

450 Answer Too Long

This status code is used by the SSC Server to inform the client that the message length for the response is to large and cannot be send.

454 Parameter Address Not Found

This status code is used by the SSC Server to inform the client that the request cannot be processed, because the requested addresses is hidden.

11.5 5xx Server Error

Response status codes beginning with the digit "5" indicate cases in which the SSC Server is aware that it has erred or is incapable of performing the request. Except when responding to a HEAD request, the SSC Server SHOULD include an entity containing an explanation of the error situation, and whether it is a temporary or permanent condition. User agents SHOULD display any included entity to the user. These response codes are applicable to any request method.

500 Internal Server Error

A generic error message, given when no more specific message is suitable.

501 Not Implemented

The SSC Server does not support the functionality required to fulfill the request. This is the appropriate response when the SSC Server does not recognize the request method and is not capable of supporting it for any resource.

503 Service Unavailable

The SSC Server is currently unable to handle the request due to a temporary overloading or maintenance of the SSC Server. The implication is that this is a temporary condition which will be alleviated after some delay.