AVBC - A PROTOCOL FOR CONNECTION MANAGEMENT AND SYSTEM CONTROL FOR AVB

V1.3 APRIL 16, 2010

Abstract

A design utilizing the OSC (Open Sound Control), HTTP and JSON protocols that can be used for any device or system to discover, enumerate, manage connections and control any AVB (Audio Video Bridging) media streams.

Keywords

AVB, AVBC, Audio Video Bridging, OSC, Open Sound Control, Digital Audio, Digital Video, Schema.

Jeff Koftinoff <jeff.koftinoff@ieee.org>

CONTENTS

CHAPTER

ONE

DISCLAIMER

Legal Disclaimer:

```
THIS DOCUMENT HAS BEEN PREPARED TO ASSIST THE IEEE P1722.1
WORKING GROUP, AND MAY BE ALTERED OR AMENDED AT A LATER DATE.
THE AUTHOR(S)S DO NOT ASSUME ANY LIABILITY IN CONNECTION WITH THE
USE OF INFORMATION OR DATA CONTAINED IN THIS DOCUMENT.
THE INFORMATION AND DATA CONTAINED IN THIS DOCUMENT ARE PROVIDED
"AS IS" WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED,
INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY,
NONINFRINGEMENT AND FITNESS FOR A PARTICULAR PURPOSE, ALL OF WHICH
ARE EXPRESSLY DISCLAIMED.
```

CHAPTER

TWO

INTRODUCTION

The Ethernet AVB standards allow for time stamped and low jitter media streams to go over ethernet links in real time between multiple devices.

The devices and streams need to be published, discovered, configured and controlled.

The media that these streams come from or go to needs to be managed.

This specification defines a simple and extensible protocol which can be used at the application level by all manufacturers of AVB devices.

2.1 Special Thanks

The following people have contributed in important discussions at face to face meetings or via email either directly or indirectly to improve this specification:

- Andy W. Schmeder <andy@cnmat.berkeley.edu>
- Adrian Freed <adrian@cnmat.berkeley.edu>
- Lee Minich <lee.minich@labxtechnologies.com>
- Chris Pane <chris.pane@labxtechnologies.com>
- Kevin B Stanton <kevin.b.stanton@intel.com>
- Jeremy Friesner <jaf@meyersound.com>
- Rob Silfvast <rob.silfvast@avid.com>
- Guy Fedorkow <guy.fedorkow@gmail.com>
- Thomas Schaefer <thomass@meyersound.com>
- Ellen Juhlin <elj@meyersound.com>
- Perrin Meyer <perrin@meyersound.com>
- Girault Jones <girault@apple.com>
- Matthew Xavier Mora <mxmora@apple.com>
- Michael Johas Teener <Mikejt@broadcom.com>

CHAPTER

THREE

OVERVIEW

This specification is for ethernet connected media oriented devices and computers which can function as AVB talkers and listeners.

All configuration, control, and response packets that a module receives or transmits are messages conforming to the Open Sound Control (OSC) specification.

These Open Sound Control messages allow for simple discovery, enumeration and configuration of:

- Individual hardware and software units called “devices” by device “name”
- Logical groups of devices on the same network called “systems”
- AVB Stream names and media properties for an **AVB Talker**
- **AVB Listener**‘s subscription of AVB streams by name
- Synchronization selection for **AVB Talker** streams
- Mapping between streams and media input/output channels
- Media input and output properties and control

This specification currently does not cover the following items:

- Ethernet port and TCP/IP configuration
- Media playback and recording streams (DVD,CD, Hard disk recorders, etc.)

However the OSC protocol schema could easily be extended to include them.

3.1 Motivation

The motivation behind this specification are the typical use case scenarios encountered by existing products in multiple markets and the need to serve these markets with pre-compliant AVB implementations in devices now.

Open Sound Control (OSC) provides a simple framework for devices to define and publish an arbitrary control structure.

This specification defines a minimal subset of control points which are useful for managing an AVB endpoint as defined by IEEE 1722 and a minimal set of control points which allows other devices to dynamically discover the required details of all of the available control points including non standard control points as needed by the unique device and market.

One of the important targets of this specification is to allow a fixed client system to query a server’s data schema and value types and limits which would allow end users to easily create and use dynamic user interfaces to control these control points of which the fixed client had no prior knowledge of.

As it is also understood that the HTTP Web Browser is the ubuquitous client application, this specification also defines an optional method to bridge javascript objects in JSON notation into OSC messages and back to allow a dynamic web application written in javascript to provide all the management tools required to configure, connect, and manage an AVB endpoint.

This specification will track the required changes to control an AVB endpoint using the IEEE 1722 protocol as the standard matures.

3.2 Reference Implementation

This specification is open and public. There will be an open source reference implementation available which would easily integrate with AVB Endpoint hardware in a microcontroller.

The open source BSD licensed reference implementation of the entire AVBC protocol in portable ANSI C code is in development and will be available at:

http://avbc.googlecode.com/

Autobuilds, autotests and continuous integration will be available at:

http://autobuild.jdkoftinoff.com/

CHAPTER

FOUR

THE OSC PROTOCOL

4.1 OSC Overview

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

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

It is a very simple and very extensible protocol that can be implemented easily in embedded systems. It can be transported over ipv4 and ipv6 protocols using UDP packets and TCP streams.

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

The OSC Schema defined by MicroOSC at:

- http://cnmat.berkeley.edu/library/uosc_project_documentation/osc_address_schema

was used as a starting point for some parts of the schema defined in this document.

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

4.2 Example OSC Implementations

oscpack: `oscpack` is a c++ implementation of the OSC protocol. `oscpack` was taken from audiomulch which was released under a BSD style open source license at http://www.audiomulch.com/~rossb/code/oscpack/ .

liblo: Lightweight OSC implementation, http://liblo.sourceforge.net/

scosc: Python interface to communicate with SuperCollider (http://www.audiosynth.com/) via OSC: http://trac2.assembla.com/pkaudio/wiki/scosc

javaosc: Java OSC library: http://www.illposed.com/software/javaosc.html

uOSC: OSC implementation for constrained PIC chips: http://cnmat.berkeley.edu/research/uosc

ScalaOSC: OSC implementation in the Scala language: http://sciss.de/scalaOSC/

4.3 OSC Transaction Syntax

The OSC Schema is described as transaction with the following syntax:

- A line prefixed with "TX" is an OSC message that some client is sending to an OSC server
- A line prefixed with "RX" is an OSC message that the OSC server will send back to the client.

Text enclosed in square brackets, "[" and "]" define a single OSC Message.

An OSC Message consists of the following in order:

- OSC Address Pattern, which defines the address of the parameter
- OSC Type Tag String, which defines the number and types of the parameter values
- OSC Data, for the values of the parameters.

For example, if a server responded to a hypothetical address "/input/level", a client sends a request for the value of an input level:

```
TX: [ "/input/level" ,i (input #) ]
RX: [ "/input/level" ,if (input #) (current value) ]
```

The server responded with the the address of the parameter, the input number, and the current value as a IEEE float.

The client sends a request for the server to set the value for an input's level to an IEEE float value:

```
TX: [ "/input/level" ,if (input #) (desired value) ]
RX: [ "/input/level" ,if (input #) (actual value) ]
```

The server responded with the level that was actually set for input 1, as an IEEE float.

4.4 The Open Sound Control 1.1 Specification

Open Sound Control (OSC) is an open, transport-independent, message-based protocol developed for communication among computers, sound synthesizers, and other multimedia devices.

4.4.1 Standard OSC Syntax

Atomic Data Types

All OSC data is composed of the following fundamental data types:

int32: 32-bit big-endian two's complement integer

OSC-timetag: 64-bit big-endian fixed-point time tag, semantics defined below

float32: 32-bit big-endian IEEE 754 floating point number

OSC-string: A sequence of non-null ASCII characters followed by a null, followed by 0-3 additional null characters to make the total number of bits a multiple of 32. (OSC-string examples) In this document, example OSC-strings will be written without the null characters, surrounded by double quotes.

OSC-blob: An int32 size count, followed by that many 8-bit bytes of arbitrary binary data, followed by 0-3 additional zero bytes to make the total number of bits a multiple of 32.

The size of every atomic data type in OSC is a multiple of 32 bits. This guarantees that if the beginning of a block of OSC data is 32-bit aligned, every number in the OSC data will be 32-bit aligned. OSC Packets

The unit of transmission of OSC is an OSC Packet. Any application that sends OSC Packets is an OSC Client; any application that receives OSC Packets is an OSC Server.

An OSC packet consists of its contents, a contiguous block of binary data, and its size, the number of 8-bit bytes that comprise the contents. The size of an OSC packet is always a multiple of 4.

The underlying network that delivers an OSC packet is responsible for delivering both the contents and the size to the OSC application. An OSC packet can be naturally represented by a datagram by a network protocol such as UDP. In a stream-based protocol such as TCP, the stream should begin with an int32 giving the size of the first packet, followed by the contents of the first packet, followed by the size of the second packet, etc.

The contents of an OSC packet must be either an OSC Message or an OSC Bundle. The first byte of the packet's contents unambiguously distinguishes between these two alternatives. OSC Messages

An OSC message consists of an OSC Address Pattern followed by an OSC Type Tag String followed by zero or more OSC Arguments.

An OSC Address Pattern is an OSC-string beginning with the character '/' (forward slash).

OSC Type Tag String

An OSC Type Tag String is an OSC-string beginning with the character ',' (comma) followed by a sequence of characters corresponding exactly to the sequence of OSC Arguments in the given message. Each character after the comma is called an OSC Type Tag and represents the type of the corresponding OSC Argument. (The requirement for OSC Type Tag Strings to start with a comma makes it easier for the recipient of an OSC Message to determine whether that OSC Message is lacking an OSC Type Tag String.)

OSC V1.1 specifies the following mandatory `Type Tags` and the type of its corresponding OSC Argument:

Type Tag	Type
b	Blob/Byte Array
F	False. No bytes are allocated in the argument data.
f	32 bit IEEE float
I	Impulse. No bytes are allocated in the argument data.
i	32 bit integer
N	Nil. No bytes are allocated in the argument data.
s	Null terminated ASCII/UTF-8 String
T	True. No bytes are allocated in the argument data.
t	OSC-timetag

Some OSC applications communicate among instances of themselves with additional, nonstandard argument types beyond those specified above. OSC applications are not required to recognize these types; an OSC application should discard any message whose OSC Type Tag String contains any unrecognized OSC Type Tags.

Extended OSC Type Tags Currently In Use

Type Tag	Type of corresponding argument
c	an ascii character, sent as 32 bits
d	64 bit ("double") IEEE 754 floating point number
h	64 bit big-endian two's complement integer
r	32 bit RGBA color
S	Alternate type represented as an OSC-string
m	4 byte MIDI message. Bytes from MSB to LSB are: port id, status byte, data1, data2
[	Indicates the beginning of an array. The tags following are for data in the Array until a close brace tag is reached.
]	Indicates the end of an array.

4.4.2 OSC Arguments

A sequence of OSC Arguments is represented by a contiguous sequence of the binary representations of each argument.

4.4.3 OSC Bundles

An OSC Bundle consists of the OSC-string "#bundle" followed by an OSC Time Tag, followed by zero or more OSC Bundle Elements. The OSC-timetag is a 64-bit fixed point time tag whose semantics are described below.

An OSC Bundle Element consists of its size and its contents. The size is an int32 representing the number of 8-bit bytes in the contents, and will always be a multiple of 4. The contents are either an OSC Message or an OSC Bundle.

Note this recursive definition: bundle may contain bundles.

This table shows the parts of a two-or-more-element OSC Bundle and the size (in 8-bit bytes) of each part.

Parts of an OSC Bundle:

Data	Size	Purpose
OSC-string "#bundle"	8 bytes	How to know that this data is a bundle
OSC-timetag	8 bytes	Time tag that applies to the entire bundle
Size of first bundle element	int32_t = 4 bytes	First bundle element
First bundle element's contents		
Size of second bundle element	int32_t = 4 bytes	Second bundle element
Second bundle element's contents		
etc.	Addtional bundle elements	

4.4.4 OSC Semantics

OSC Address Spaces and OSC Addresses

Every OSC server has a set of OSC Methods. OSC Methods are the potential destinations of OSC messages received by the OSC server and correspond to each of the points of control that the application makes available. "Invoking" an OSC method is analogous to a procedure call; it means supplying the method with arguments and causing the method's effect to take place.

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

Each OSC Method and each OSC Container other than the root of the tree has a symbolic name, an ASCII string consiting of printable characters other than the following:

Printable ASCII characters not allowed in names of OSC Methods or OSC Containers

character	name	ASCII code (decimal)
' '	space	32
#	number sign	35
*	asterisk	42
,	comma	44
/	forward slash	47
//	double forward slash	47,47
?	question mark	63
[	open bracket	91
]	close bracket	93
{	open curly brace	123
}	close curly brace	125

oThe OSC Address of an OSC Method is a symbolic name giving the full path to the OSC Method in the OSC Address Space, starting from the root of the tree. An OSC Method's OSC 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 OSC Method, separated by forward slash characters, followed by the name of the OSC Method. The syntax of OSC Addresses was chosen to match the syntax of URLs.

OSC Message Dispatching and Pattern Matching

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

The parts of an OSC Address or an OSC Address Pattern are the substrings between adjacent pairs of forward slash characters and the substring after the last forward slash character.

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

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

However, if the double forward slash is used in the OSC Address Pattern, then:

1. The "//" can match multiple levels of parts.

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

1. '?' in the OSC Address Pattern matches any single character
2. '*' in the OSC Address Pattern matches any sequence of zero or more characters
3. A string of characters in square brackets (e.g., "`[string]`") in the OSC Address Pattern matches any character in the string. Inside square brackets, the minus sign (`-`) and exclamation point (`!`) have special meanings:

- Two characters separated by a minus sign indicate the range of characters between the given two in ASCII collating sequence. A minus sign at the end of the string has no special meaning.
- An exclamation point at the beginning of a bracketed string negates the sense of the list, meaning that the list matches any character not in the list. (An exclamation point anywhere besides the first character after the open bracket has no special meaning.)

4. A comma-separated list of strings enclosed in curly braces (e.g., "{foo,bar}") in the OSC Address Pattern matches any of the strings in the list.
5. The "//" pattern matches any strings between '/' characters, including other '/' characters.
6. Any other character in an OSC Address Pattern can match only the same character.

4.4.5 Temporal Semantics and OSC Time Tags

An OSC server must have access to a representation of the correct current absolute time. OSC does not provide any mechanism for clock synchronization.

When a received OSC Packet contains only a single OSC Message, the OSC Server should invoke the correponding OSC Methods immediately, i.e., as soon as possible after receipt of the packet. Otherwise a received OSC Packet contains an OSC Bundle, in which case the OSC Bundle's OSC Time Tag determines when the OSC Bundle's OSC Messages' corresponding OSC Methods should be invoked. If the time represented by the OSC Time Tag is before or

equal to the current time, the OSC Server should invoke the methods immediately (unless the user has configured the OSC Server to discard messages that arrive too late). Otherwise the OSC Time Tag represents a time in the future, and the OSC server must store the OSC Bundle until the specified time and then invoke the appropriate OSC Methods.

Time tags are represented by a 64 bit fixed point number. The first 32 bits specify the number of seconds since midnight on January 1, 1900, and the last 32 bits specify fractional parts of a second to a precision of about 200 picoseconds. This is the representation used by Internet NTP timestamps. The time tag value consisting of 63 zero bits followed by a one in the least signifigant bit is a special case meaning "immediately."

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

When an OSC Address Pattern is dispatched to multiple OSC Methods, the order in which the matching OSC Methods are invoked is unspecified. When an OSC Bundle contains multiple OSC Messages, the sets of OSC Methods corresponding to the OSC Messages must be invoked in the same order as the OSC Messages appear in the packet.

When bundles contain other bundles, the OSC Time Tag of the enclosed bundle must be greater than or equal to the OSC Time Tag of the enclosing bundle. The atomicity requirement for OSC Messages in the same OSC Bundle does not apply to OSC Bundles within an OSC Bundle.

4.5 Open Sound Control Extensions

The Open Sound Control standards define the data format of an OSC message or bundle on the wire and a simple addressing scheme. These definitions are flexible enough to allow us to extend the semantics of OSC to include the higher level messaging constructs:

- Ping
- Error reporting
- Schema reflection
- Value limits reflection
- Subscriptions

4.5.1 Ping

When using a TCP/IP stream, it is often useful for the client to send a request over the socket to make sure that the socket and server is still active and working and did not silently disappear without an orderly socket shutdown.

The `"/osc/ping"` address is used for this. When a client sends a `"/osc/ping"` request the server will immediately respond with an `"/osc/pong"` response.

4.5.2 Error Reporting

The server shall respond to all requests that generate an error with an "/osc/error" message in the following format:

```
RX: [ "/osc/error" ,iss... (error code) (error description) (original request address)
      (original request values) ]
```

The error code is defined in the spirit of HTTP response codes.

The error code is a three digit integer. The error description is an optional textual description in case further information is available by the server. It may be ommitted by using a `'N'` typetag.

The first digit of the error code defines the class of response. The last two digits do not have any categorization role. There are 4 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 client device would only have to look at the first digit of the error code in order to determine what options are available to deal with the response. A more complex device could use the more detail in the two digits or the description text in order to help the end user understand what happened.

1xx - Informational response

The 1xx responses would typically be used during debugging or tracing during development.

Code	Description	Detail
100	Trace Message	A single step in multi-step process was completed
101	Debug Message	Arbitrary debugging information was sent

2xx - Success

The 2xx responses signify that either a value was correctly set or a resource was aquired successfully.

Code	Description	Detail
200	OK	The command was executed successfully
201	Created	A resource was created successfully
202	Started	A request to start a task was accepted and the task started
203	Deleted	A resource was deleted successfully
204	Stopped	A request to stop a task was accepted and the task stopped

3xx - Incomplete

The 3xx response is used to tell the client that the request requires further action before it can complete.

Code	Description	Detail
300	Incomplete	Incomplete

4xx - Client Error

The 4xx response is used to tell the client that the request was invalid in some way.

Code	Description	Detail
400	Bad Address	The address was not recognized or matched
401	Bad Type	Typetag(s) selected was not recognized
402	Bad Parameter	Parameter(s) for the address were invalid
403	Bad Value	Values(s) for the address were out of range
404	Auth Required	Authentication is required
405	Time Tag Past	A Bundle's Time Tag was missed
406	Time Tag Future	A Bundle's Time Tag was too far in the future

5xx - Server Error

The 5xx response is used to tell the client that the request seemed valid but the server was unable to complete the request.

Code	Description	Detail
500	Internal Error	Some unclassifiable internal error occurred
501	Not Implemented	The request relies on a feature in the server that is not implemented
502	Time Out	Some internal or external process had a transitory failure. Retrying may succeed.
503	Service Unavailable	The system is not ready to handle requests at this time.

4.5.3 Schema reflection

The AVBC protocol defines methods for a client to query the currently active schema information for any address by prefixing the address with `'/osc/schema'` and sending a request for the values. The response to this query are individual strings listing each of the items or address containers within the specified address. Address containers are suffixed by the `'/'` character. Using `'/osc/schema'`, the client can walk all addresses available within a device's tree if needed.

4.5.4 Value limits reflection

The AVBC protocol defines methods for a client to query the currently active value limits information by prefixing the address with `'/osc/limits'` and sending a request for the values. The response to this query are arrays of key/value pairs defining the various properties that the values for an address may contain.

4.5.5 Subscriptions

A subscription request is sent by a client to a server for an address pattern to subscribe to, along with subscription throttling properties. Subscriptions made via a TCP/IP socket are sent back on the same socket. Subscriptions made via a unicast or multicast or broadcast UDP packet are sent to the client Sender IP address and Sender Port via Unicast. A future revision of this specification may include the ability for the client to request the server to transmit subscription values to a multicast group.

4.6 TCP/IP Transport of AVBC

A device implementing the AVBC protocol can function as an "AVBC Protocol Client" or as an "AVBC Protocol Server", or both at the same time.

4.6.1 AVBC Protocol Client

A device functioning as an "AVBC Protocol Client":

- MUST be able to send and receive OSC Messages and OSC Bundles via IPv4 UDP messages.
- MUST be able to send IPv4 UDP messages to an IPv4 Multicast Address.
- MAY support TCP streams for OSC Message/Bundle transport.
- MAY support IPv6 unicast and multicast.

4.6.2 AVBC Protocol Server

A device functioning as an "AVBC Protocol Server":

- MUST listen to UDP port 17220
- MUST be able to send and receive OSC Messages and OSC Bundles via IPv4 UDP messages.
- MUST be able to join an IPv4 Multicast Address via IGMP in order to receive the multicast IPv4 UDP messages.
- MAY support IPv6 unicast and multicast.
- MUST support TCP streams for OSC Message and OSC Bundle transport.

IPv4 and IPv6 multicast addresses have not been allocated for this protocol. A future revision of AVBC may specify that a special OSC address could to be used to tell a specific client device which multicast addresses to use to join to listen.

4.6.3 TCP/IP Streams Packet Encoding

When a device transmits a message via a TCP stream, the message is "SLIP Frame Encoded" as per the OSC v1.1 specification.

Because the responses to a AVBC packet contain all the information that was in the original request, both requests and responses can and should be pipelined.

4.6.4 UDP Packet Transmission and Responses

When a client transmits a message via a UDP packet, the UDP packet's source port MUST be the same port as the device listens to incoming packets on. This requirement allows multiple clients to live on one IP address, such as different programs on a desktop computer, and each client can send a multicast or broadcast UDP packet to the network and get their own responses back. When an AVB Protocol Server receives the message and has a response, it MUST respond to the message via a unicast message addressed to the source IP address and port from the request.

4.6.5 Client Requests

Clients initiate requests to servers. A client can:

- Send a request to a server via a unicast UDP message. If the server has a response for the client the server sends it via unicast UDP to the source IP address and source port of the original UDP message.
- Send a request to a server via a multicast UDP message. If the server has a response for the client the server sends it via unicast UDP to the source IP address and source port of the original UDP message.
- Send a request to a server via a TCP stream. If the server has a response for the client the server sends it via the same TCP socket connection. The server will not close the socket until the client closes the socket. Client requests to the `"/osc/ping"` will be responded with an `"/osc/pong"` response and can be used to make sure the connection is still alive for TCP and UDP.

CHAPTER

FIVE

AVB OSC SCHEMA

OSC is extensible. Any device manufacturer is free to add new addresses and even new type tags to the protocol as they see fit, as long as the base standard is adhered to.

There is however a minimal OSC schema required for AVB devices which would allow basic interoperability as well as discovery of extended features. This schema includes:

5.1 OSC Meta-Information - `/osc/`

All devices respond to the `/osc/` container to allow for clients to query OSC protocol version and capabilities.

These are roughly based on the MicroOSC address schema available at:

- http://cnmat.berkeley.edu/library/uosc_project_documentation/osc_address_schema/osc

5.1.1 `/osc/version`

Read only value.

Report the OSC Version implemented in the server:

```
TX: [ "/osc/version" ]
RX: [ "/osc/version" ,s "1.1" ]
```

5.1.2 `/osc/type/accepts` and `/osc/type/reports`

Read only values.

List type tags understood by device and transmitted by the device:

```
TX: [ "/osc/type/accepts" ]
RX: [ "/osc/type/accepts" ,s "ifsbhTFI" ]

TX: [ "/osc/type/reports" ]
RX: [ "/osc/type/reports" ,s "ifsbhTFI" ]
```

Each character in the string value is an OSC type tag; for instance in this example:

- "`i`" means *32 bit integers*
- "`f`" means *IEEE 32 bit floats*

- "s" means *strings*
- "b" means *blobs*
- "h" means *64 bit integers*
- "T" means *TRUE*
- "F" means *FALSE*
- "I" means *Impulse*

5.1.3 Ping/Pong

When a client sends a "/osc/ping" request the server will immediately respond with an "/osc/pong" response with the identical parameters. With no parameters:

```
TX: [ "/osc/ping" ]
RX: [ "/osc/pong" ]
```

With some parameters:

```
TX: [ "/osc/ping" ,ssif "foo" "bar" 42 123.456 ]
RX: [ "/osc/pong" ,ssif "foo" "bar" 42 123.456 ]
```

This can be used by simple clients to assist in sequencing pipelines.

5.1.4 The Schema Predicate - /osc/schema

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

For instance the following query on the top level "/avb/" address:

```
TX: [ "/osc/schema/media/" ]
```

Would return the following OSC Message which describes the addresses that are contained in the "/media/" address one level deep:

```
RX: [ "/osc/schema/media/" ,ssss "ins" "in/" "outs" "out/" ]
```

Note that the responses end with a "/" character if that address is a container for more addresses.

And the following query could then be performed to query the "/avb/source" container:

```
TX: [ "/osc/schema/avb/source" ]
```

And it could return the following OSC Message in the case of a device with four active avb stream sources, assuming that the MAC prefix of this device was the 48 bit value "12345678abcd":

```
RX: [ "/osc/schema/media/source" , "formats" "create" "byname/"
      "12345678abcd0001" "12345678abcd0002"
      "12345678abcd0003" "12345678abcd0004" ]
```

5.1.5 The Limits Predicate - `/osc/limits`

The `/osc/limits` predicate can be prepended to any other OSC request. The response of the request is always a collection of pairs of parameters describing properties of the addressed object.

The returned parameters for `/osc/limits` always come in arrays of key/value pairs describing each property of a value in an address. One array is used for each value item in an address.

The first item of a pair is a string description of the property type. The second item of a pair is a value for that property.

This property list is extensible in a future revision of this protocol standard.

Currently defined optional properties are:

Property name	Value	Description
`type`	string	typetags acceptable for this address
`min`	numeric	minimum value
`max`	numeric	maximum value
`inc`	numeric	recommended user interface increment value
`units`	string	string describing value units (SI)
`description`	string	optional string describing this address
`option`	array	individual option values for this address

Example #1 of `/osc/limits` usage

For the optional address for media sink 1's level:

```
TX: [ "/media/sink/1/level" ]
RX: [ "/media/sink/1/level" ,f (level in dB) ]
```

To request the limits for "/media/sink/1/level":

```
TX: [ "/osc/limits/media/sink/1/level" ]
RX: [ "/osc/limits/media/sink/1/level" ,[sssfsfsfss]
      "type" "f" "min" -100.0 "max" 10.0
      "inc" 0.1 "units" "dB" ]
```

The response of the limits request shows that this address/parameter combination accepts a single value in `float32` format. The units are "dB" (decibels) and the minimum value is -100.0 dB and the maximum value is 10.0 dB. The prefered increment for these values is 0.1 dB.

Example #2 of `/osc/limits` usage

Take the fictituous vendor specific address for a media source's scale options on a device that has a `[VENDOR_OUI]` of `123456`:

```
TX: [ "/media/source/1/vendor/123456/scale" ]
RX: [ "/media/source/1/vendor/123456/scale" ,i (scale code) ]
```

And this device allowed the only valid values of "-10", "0" and "16" dB.

To request the limits for "/media/source/1/vendor/123456/scale":

```
TX: [ "/osc/limits/media/source/1/vendor/123456/scale" ]
RX: [ "/osc/limits/media/source/1/vendor/123456/scale" ,[sss[iii]ss]
      "type" "i"
      "option" -10 0 16
      "units" "dB" ]
```

The response of the limits request shows that this address/parameter combination accepts a single value in `int32` format. The units are "dB" (decibels) and the only values allowed are -10, 0, and 16.

By using the "/osc/schema" and "/osc/limits" commands repeatedly a client can discover all of the available parameters exposed via OSC that are available in a module.

5.1.6 Subscriptions

A subscription request is sent by a client to a server for an address pattern to subscribe to. Subscriptions always expire in 10 seconds and must be renewed before this timeout is complete if the client wishes to continue to subscribe. The server shall send all state changes to the client. The frequency and throttling of the state updates is determined by the following optional subscription properties:

Subscription Property Name	Default Value	Description
`"min"`	100 ms	Minimum update period (0=none)
`"max"`	1000 ms	Maximum update period (0=none)
`"bw"`	100000 bytes/sec	Maximum bandwidth to send

Additional details:

- If `"min"` is 0, then updates are not sent when a value changes, they are only sent based on the `"max"` period.
- If `"max"` is 0, then updates are only sent when a value changes based on the `"min"` period.

The format of a subscription request is:

```
TX: [ "/osc/state/subscribe" ,s[si..]
      (address pattern to subscribe to)
      (property name) (property value)
      [repeat propery name/value]
      ]
```

After 3 seconds, the client should re-send the request if the client is still interested in receiving this subscription for another 10 seconds. This allows a subscription renew request UDP packet to be lost twice in a row without interrupting the subscription.

Subscription Example #1

A client is interested in the state of all AVB stream sources coming from a single device. It wants updates of changes no faster than 100 ms per address and it wants all address values to be sent every 2000 ms. It does not want to receive more than 100000 bytes per second from this subscription.

It connects to a server and subscribe to the `"/avb/source/*/state"` address pattern and sends the following request:

```
TX: [ "/osc/state/subscribe" ,s[sisisi]
      "/avb/source/*/state"
      "min" 100
      "max" 2000
      "bw"  100000 ]
```

```
RX: [ "/osc/state/subscribe" ,s[sisisi]
      "/avb/source/*/state"
      "min" 100
      "max" 2000
      "bw"  100000 ]
```

Because these property values are the same as the defaults, the client could send the following instead:

```
TX: [ "/osc/state/subscribe" ,s "/avb/source/*/state" ]
```

The server will send the values to the client when they change. If a value changes more than once or twice within 100 ms, value changes will be dropped and the most recent value will be sent. If a value does not change within 2000 ms, the value will still be sent once per 2000 ms. If the data rate of this subscription exceeds 100000 bytes per second, transmissions will be throttled.

Subscription Example #2

A client is interested in all changes and state in the entire `"/avb/"` and `"/media/"` trees.:

```
TX: [ "/osc/state/subscribe" ,s "/avb//" ]
RX: [ "/osc/state/subscribe" ,s "/avb//" ]
TX: [ "/osc/state/subscribe" ,s "/media//" ]
RX: [ "/osc/state/subscribe" ,s "/media//" ]
```

The server will notify the client whenever any state changes in any of the trees at any level.

5.2 Device Settings - `/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/vendor_id`

The address, `/device/identity/vendor_id` is used as a read only value that contains an *int32* containing the vendor's 24 bit OUI code.

Getting device vendor:

```
TX: [ "/device/identity/vendor_id" ]
RX: [ "/device/identity/vendor_id" ,s (vendor) ]
```

5.2.2 `/device/identity/vendor`

The address, `/device/identity/vendor` is used as a read only value that contains the following information:

- A *string* containing the vendor name in UTF-8

Getting device vendor:

```
TX: [ "/device/identity/vendor" ]
RX: [ "/device/identity/vendor" ,s (vendor) ]
```

5.2.3 /device/identity/product

The address, `/device/identity/product` is used as a read only value that contains the following information:

- A *string* containing the product name

Getting device product name:

```
TX: [ "/device/identity/product" ]
RX: [ "/device/identity/product" ,s (product) ]
```

5.2.4 /device/identity/version

The address, `/device/identity/version` is used as a read only value that contains the following information:

- A single *string* or multiple *strings* containing the product's firmware/software/hardware version

Getting device versions:

```
TX: [ "/device/identity/version" ]
RX: [ "/device/identity/version" ,sss (software version) (hardware version) ]
```

5.2.5 /device/identity/serial

The address, `/device/identity/serial` is used as a read only value that contains the following information:

- A *string* containing the product serial number

A serial number is expected to be unique within this vendor/product combination. If the device does not have a unique serial number, the MAC address of the first ethernet port on the device may be used.

Getting device product serial number:

```
TX: [ "/device/identity/serial" ]
RX: [ "/device/identity/serial" ,s (serial) ]
```

5.2.6 /device/name

The `/device/name` address is used as an end-user configurable name for the device. This device name will be used to set the bonjour device host name. This device name may be automatically changed by Bonjour if the requested name is not unique on the network.

Read/write value.

Getting device name:

```
TX: [ "/device/name" ]
RX: [ "/device/name" ,s (value) ]
```

Setting device name:

```
TX: [ "/device/name" ,s (value) ]
RX: [ "/device/name" ,s (value) ]
```

5.2.7 /device/system

Read/write value.

The system name is the system that this device is a member of.

Getting System Name:

```
TX: [ "/device/system" ]
RX: [ "/device/system" ,s (value) ]
```

Setting System Name:

```
TX: [ "/device/system" ,s (value) ]
RX: [ "/device/system" ,s (value) ]
```

5.3 AVB Control - /avb/

5.3.1 AVB Stream Sources

/avb/source/formats

Read only value.

Get a list of all the 1722 formats that are supported by the avb endpoint as a stream source:

```
TX: [ "/avb/source/formats" ]
RX: [ "/avb/source/formats" ,s... (values)... ]
```

The format string used for various media formats is TBD. It will map directly to 1722 and IEC61883.

/avb/source/create

Create an AVB stream source:

```
TX: [ "/avb/source/create" ,NNsshii... (stream name) (media format)
      (presentation) (number of channels) (media source channel code)... ]
RX: [ "/avb/source/create" ,hhsshii... (stream id) (multicast MAC
      address) (stream name) (media format) (presentation) (number of channels)
      (media source channel code)... ]
```

If the requested stream is already successfully created and the client requests a stream source to be created with the exact same name and parameters, then the server will respond with the existing stream information.

Stream source names must be unique within a single talker device.

The channel map information is defined as a list of “media source channel codes”. A “media source channel code” is a split int32. The upper 16 bits defines a media source id. The lower 16 bits defines a channel within that media source.

Create AVB Stream Source Example #1

A talker device has some AES3 digital audio inputs designated as stereo media sources #1,#2 and #3. The client wants to take the audio from these AES3 channels and create a single 6 channel audio stream suitable for a 5.1 audio format with the default presentation time.:

```
TX: [ "/avb/source/create" ,NNssNiiiiiii "Movie Audio" "(Format codes TBD)"
      6 0x00010000 0x00010001 0x00020000 0x00020001 0x00030000 0x0003001 ]
RX: [ "/avb/source/create" ,hhsshiiiiiii (stream id) (multicast MAC address)
      "Movie Audio" "(Format codes TBD)" (presentation time offset in ns)
      6 0x00010000 0x00010001 0x00020000 0x00020001 0x00030000 0x0003001 ]
```

/avb/source/byname/[STREAM_NAME]/

Alias to the container `/avb/source/[STREAM_ID]/` that matches the specified `[STREAM_NAME]`.

/avb/source/[STREAM_ID]/

Container for the specified stream source by id. The stream id here includes the 48 bit ethernet MAC Address that it is sent to as well as the 16 bit stream number. The value is a 64 bit integer.

/avb/source/[STREAM_ID]/destroy

Destroy the specified stream source:

```
TX: [ "/avb/source/[STREAM_ID]/destroy" ,I ]
RX: [ "/avb/source/[STREAM_ID]/destroy" ,I ]
```

/avb/source/[STREAM_ID]/id

This read only value specifies the full stream ID for this stream.

Request current stream id:

```
TX: [ "/avb/source/[STREAM_ID]/id" ]
RX: [ "/avb/source/[STREAM_ID]/id" ,h (mac and id) ]
```

/avb/source/[STREAM_ID]/mmac

This read only value specifies the current multicast mac address used for this stream id.

Request current stream multicast mac address:

```
TX: [ "/avb/source/[STREAM_ID]/mmac" ]
RX: [ "/avb/source/[STREAM_ID]/mmac" ,h (mac and id) ]
```

/avb/source/[STREAM_ID]/state

Read only value

Get current state of this stream source:

```
TX: [ "/avb/source/[STREAM_ID]/state" ]
RX: [ "/avb/source/[STREAM_ID]/state" ,s "potential" ]
```

Valid stream states are:

- "potential"
- "reserved"
- "active"

/avb/source/[STREAM_ID]/format

This read/write vaue specifies which format the talker for this stream is transmitting

Request current format:

```
TX: [ "/avb/source/[STREAM_ID]/format" ]
RX: [ "/avb/source/[STREAM_ID]/format" ,s (format) ]
```

Set the transmission format:

```
TX: [ "/avb/source/[STREAM_ID]/format" ,s (new format) ]
RX: [ "/avb/source/[STREAM_ID]/format" ,s (new format) ]
```

The format string used for various media formats is TBD. It will map directly to 1722.1 and IEC61883.

/avb/source/[STREAM_ID]/channels

Request current channel count for stream:

```
TX: [ "/avb/source/[STREAM_ID]/channels" ]
RX: [ "/avb/source/[STREAM_ID]/channels" ,i (channels) ]
```

/avb/source/[STREAM_ID]/map

Request the current channel map for the Stream Sink:

```
TX: [ "/avb/source/[STREAM_ID]/map" ]
RX: [ "/avb/source/[STREAM_ID]/map" ,i... (media channel source code)... ]
```

Set expected channel map for stream:

```
TX: [ "/avb/source/[STREAM_ID]/map" ,i... (media channel source code)... ]
RX: [ "/avb/source/[STREAM_ID]/map" ,i... (media channel source code)... ]
```

/avb/source/[STREAM_ID]/presentation

This read/write value specifies the presentation time offset for the stream. The standard default AVB presentation time offset is 2.0 milliseconds.

In special situations, fine-tuning these presentation synchronization may be necessary.

Request current presentation time offset:

```
TX: [ "/avb/source/[STREAM_ID]/presentation" ]
RX: [ "/avb/source/[STREAM_ID]/presentation" ,i (time in nanoseconds) ]
```

Set the talker's presentation time offset:

```
TX: [ "/avb/source/[STREAM_ID]/presentation" ,i (time in nanoseconds) ]
RX: [ "/avb/source/[STREAM_ID]/presentation" ,i (time in nanoseconds) ]
```

All talker stream presentation time offset values default to 2,000,000 nanoseconds

5.3.2 AVB Stream Sinks

/avb/sink/formats

Read only value.

Get a list of all the 1722 formats that are supported for stream sinks:

```
TX: [ "/avb/sink/formats" ]
RX: [ "/avb/sink/formats" ,s... (values)... ]
```

The format string used for various media formats is TBD. It will map directly to 1722.1 and IEC61883.

/avb/sink/create

The `/avb/sink/create` address is used to create an AVB stream sink. It may be used in a number of ways. It can create an AVB stream sink:

- By specifying a stream ID and multicast MAC address
- By specifying a stream ID and looking up the multicast MAC address
- By specifying a device name and stream name and looking up the device IP address and looking up the stream ID and multicast MAC address from that device.

Create an AVB stream sink by specifying a stream id:

```
TX: [ "/avb/sink/create" ,NNNhNhii... (stream id)
      (presentation) (number of channels) (media sink channel code)... ]
RX: [ "/avb/sink/create" ,NNhhhhii... (sink id) (stream id) (multicast mac address)
      (presentation) (number of channels) (media sink channel code)... ]
```

Create an AVB stream sink by specifying a stream id and multicast mac address:

```
TX: [ "/avb/sink/create" ,NNNhhhii... (stream id) (multicast mac address)
      (presentation) (number of channels) (media sink channel code)... ]
RX: [ "/avb/sink/create" ,NNhhhhii...(sink id) (stream id) (multicast mac address)
      (presentation) (number of channels) (media sink channel code)... ]
```

Create an AVB stream sink by specifying a device name and stream name:

```
TX: [ "/avb/sink/create" ,ssNNhii... (device name) (stream name) (presentation)
      (number of channels) (channel map)... ]
RX: [ "/avb/sink/create" ,sshhhhii... (device name) (stream name) (sink id) (stream id)
      (multicast mac address) (presentation) (number of channels) (channel map)... ]
```

The channel map information is defined as a list of "media sink channel codes". A "media sink channel code" is a split int32. The upper 16 bits defines a media sink id. The lower 16 bits defines a channel within that media sink.

/avb/sink/[SINK_ID]/destroy

Destroy the specified stream sink:

```
TX: [ "/avb/sink/[SINK_ID]/destroy" ,T ]
RX: [ "/avb/sink/[SINK_ID]/destroy" ,T ]
```

/avb/sink/[SINK_ID]/id

This read only value specifies the full sink ID for this sink. The sink id here includes the 48 bit ethernet MAC Address that it is sent to as well as the 16 bit sink number. The value is a 64 bit integer.

Request current stream id:

```
TX: [ "/avb/sink/[SINK_ID]/id" ]
RX: [ "/avb/sink/[SINK_ID]/id" ,h (mac and id) ]
```

/avb/sink/[SINK_ID]/format

This read only vaue specifies which format the listener for this stream is currently receiving.

Request current format:

```
TX: [ "/avb/sink/[SINK_ID]/format" ]
RX: [ "/avb/sink/[SINK_ID]/format" ,s (format) ]
```

The format string used for various media formats is TBD. It will map directly to 1722.1 and IEC61883.

/avb/sink/[SINK_ID]/channels

Request channel count for stream:

```
TX: [ "/avb/sink/[SINK_ID]/channels" ]
RX: [ "/avb/sink/[SINK_ID]/channels" ,i (channels) ]
```

/avb/sink/[SINK_ID]/map

Request the current channel map for the Stream Sink:

```
TX: [ "/avb/sink/[SINK_ID]/map" ]
RX: [ "/avb/sink/[SINK_ID]/map" ,i... (media sink channel code)... ]
```

Set expected channel map for stream:

```
TX: [ "/avb/sink/[SINK_ID]/map" ,i... (media sink channel code)... ]
RX: [ "/avb/sink/[SINK_ID]/map" ,i... (media sink channel code)... ]
```

/avb/sink/[SINK_ID]/state

Read only value

Get current state of this stream source:

```
TX: [ "/avb/sink/[STREAM_ID]/state" ]
RX: [ "/avb/sink/[STREAM_ID]/state" ,s (stream state) ]
```

Valid stream states are:

- “unknown”
- “pending”
- “active”

/avb/sink/[SINK_ID]/source/device

Device name of source device for this stream sink:

```
TX: [ "/avb/sink/[SINK_ID]/source/device" ]
RX: [ "/avb/sink/[SINK_ID]/source/device" ,s (device name) ]
```

/avb/sink/[SINK_ID]/source/name

Stream name on source device for this stream sink:

```
TX: [ "/avb/sink/[SINK_ID]/source/name" ]
RX: [ "/avb/sink/[SINK_ID]/source/name" ,s (stream name) ]
```

/avb/sink/[SINK_ID]/source/id

Stream ID of source for this stream sink:

```
TX: [ "/avb/sink/[SINK_ID]/source/id" ]
RX: [ "/avb/sink/[SINK_ID]/source/id" ,h (stream id) ]
```

/avb/sink/[SINK_ID]/source/mmac

Multicast mac address of source for this stream sink:

```
TX: [ "/avb/sink/[SINK_ID]/source/mmac" ]
RX: [ "/avb/sink/[SINK_ID]/source/mmac" ,h (multicast mac) ]
```

/avb/sink/[SINK_ID]/presentation

This read/write value specifies the presentation time offset for the stream.

In special situations, fine-tuning these presentation synchronization may be necessary.

Request current presentation time offset:

```
TX: [ "/avb/sink/[SINK_ID]/presentation" ]
RX: [ "/avb/sink/[SINK_ID]/presentation" ,i (time in nanoseconds) ]
```

Set the sink's presentation time offset:

```
TX: [ "/avb/sink/[SINK_ID]/presentation" ,i (time in nanoseconds) ]
RX: [ "/avb/sink/[SINK_ID]/presentation" ,i (time in nanoseconds) ]
```

All stream presentation time offset values default to 0 nanoseconds.

5.4 Media Source Control - /media/source/

A "Media Source" is defined as a single channel or a logical collection of channels of audio or a combined audio/video stream that can be sent in an AVB Stream.

This means that a "Media Source" can be things such as:

- A single microphone input or analog line input
- A pair of related audio inputs such as the left and right signals from an external CD player
- A pair of inputs from an SPDIF or AES3 digital audio input
- A single audio playback channel from a hard disk recorder/playback system
- A number of related audio playback channels from a hard disk recorder/playback system
- A combined audio and video stream such as MPEG4 which may have multiple audio channels as well as a video channel

In these OSC addresses, `[MEDIA_SOURCE_ID]` used here represents some unique integer identifier for a media source on this device.

The following addresses are required to be available:

- `/media/source/[MEDIA_SOURCE_ID]/type`
- `/media/source/[MEDIA_SOURCE_ID]/description`
- `/media/source/[MEDIA_SOURCE_ID]/channels`

The following addresses are defined but are optional depending on the ability or feature set of the device:

- `/media/source/[MEDIA_SOURCE_ID]/mute`

- /media/source/[MEDIA_SOURCE_ID]/pan
- /media/source/[MEDIA_SOURCE_ID]/level
- /media/source/[MEDIA_SOURCE_ID]/playstate
- /media/source/[MEDIA_SOURCE_ID]/play
- /media/source/[MEDIA_SOURCE_ID]/pause
- /media/source/[MEDIA_SOURCE_ID]/stop
- /media/source/[MEDIA_SOURCE_ID]/tracks
- /media/source/[MEDIA_SOURCE_ID]/track
- /media/source/[MEDIA_SOURCE_ID]/position
- /media/source/[MEDIA_SOURCE_ID]/ff
- /media/source/[MEDIA_SOURCE_ID]/rw
- /media/source/[MEDIA_SOURCE_ID]/scanfwd
- /media/source/[MEDIA_SOURCE_ID]/scanback

The organization of these addresses is such that even an open ended simple user interface could easily provide push buttons and knobs for these common functions to the end user.

The implementor can add any other addresses as necessary for the device in the optional "vendor" container. Examples of additional addresses could items such as:

- /media/source/[MEDIA_SOURCE_ID]/vendor/[VENDOR_OUI]/meter
- /media/source/[MEDIA_SOURCE_ID]/vendor/[VENDOR_OUI]/phantom
- /media/source/[MEDIA_SOURCE_ID]/vendor/[VENDOR_OUI]/scale

These additional addresses would not need to fit any specific form or function. Since they are discoverable via the /osc/schema and /osc/limits addresses any client can know how to represent these additional addresses to the end user.

5.4.1 /media/source/[MEDIA_SOURCE_ID]/id

Readonly value.

Get the current identifier for the specified media source:

```
TX: [ "/media/source/[MEDIA_SOURCE_ID]/id" ]
RX: [ "/media/source/[MEDIA_SOURCE_ID]/id" ,i 1 ]
```

5.4.2 /media/source/[MEDIA_SOURCE_ID]/type

Read only value.

Returns the user readable ASCII type of the specified media source.

Reading:

```
TX: [ "/media/source/[MEDIA_SOURCE_ID]/type" ]
RX: [ "/media/source/[MEDIA_SOURCE_ID]/type" ,s (media type string) ]
```

Examples of media type strings could be:

- "Analog Microphone Audio Input"
- "Analog Line In Audio Input"
- "AES/EBU Digital Input L"
- "AES/EBU Digital Input R"

note: This may need to be enumerated.

5.4.3 /media/source/[MEDIA_SOURCE_ID]/description

Read only value.

Returns the user readable ASCII description of the specified media source.

Reading:

```
TX: [ "/media/source/[MEDIA_SOURCE_ID]/description" ]
RX: [ "/media/source/[MEDIA_SOURCE_ID]/description" ,s (media description string) ]
```

Examples of media type strings could be:

- "Back Panel Microphone Input #1"
- "Hard Disk Playback Channel #5"
- "MIDI Input Stream"
- "SMPTE LTC Input"

5.4.4 /media/source/[MEDIA_SOURCE_ID]/channels

Read only value.

Returns the count of media channels for this media source.

Reading:

```
TX: [ "/media/source/[MEDIA_SOURCE_ID]/channels" ]
RX: [ "/media/source/[MEDIA_SOURCE_ID]/channels" ,i (media source channel count) ]
```

5.4.5 /media/source/[MEDIA_SOURCE_ID]/mute

Optional read/write boolean value for audio channels.

A typetag of 'T' means the channel is muted. A typetag of 'F' means the channel is not muted. Changing a mute value does not change the media source's level value.

Get the current mute setting for the specified media source:

```
TX: [ "/media/source/[MEDIA_SOURCE_ID]/mute" ]
RX: [ "/media/source/[MEDIA_SOURCE_ID]/mute" ,T ]
or:
RX: [ "/media/source/[MEDIA_SOURCE_ID]/mute" ,F ]
```

Set the media source to be muted:

```
TX: [ "/media/source/[MEDIA_SOURCE_ID]/mute" ,T ]
RX: [ "/media/source/[MEDIA_SOURCE_ID]/mute" ,T ]
```

Set the media source to be unmuted:

```
TX: [ "/media/source/[MEDIA_SOURCE_ID]/mute" ,F ]
RX: [ "/media/source/[MEDIA_SOURCE_ID]/mute" ,F ]
```

5.4.6 /media/source/[MEDIA_SOURCE_ID]/pan

Optional read/write float32 value for media panpot from -1.0 to +1.0

Get the current pan setting for the specified media source.:

```
TX: [ "/media/source/[MEDIA_SOURCE_ID]/pan" ]
RX: [ "/media/source/[MEDIA_SOURCE_ID]/pan" ,f (value) ]
```

Set the current pan setting:

```
TX: [ "/media/source/[MEDIA_SOURCE_ID]/pan" ,f (value) ]
RX: [ "/media/source/[MEDIA_SOURCE_ID]/pan" ,f (value) ]
```

For media sources that have more than two channels, the pan setting may respond to multiple values/dimensions.

5.4.7 /media/source/[MEDIA_SOURCE_ID]/level

Optional read/write float32 value for media source level in dB. Changing a level value does not change the media source's mute value.

Get the current level setting for the specified media source.:

```
TX: [ "/media/source/[MEDIA_SOURCE_ID]/level" ]
RX: [ "/media/source/[MEDIA_SOURCE_ID]/level" ,f (level in dB) ]
```

Set the media source level:

```
TX: [ "/media/source/[MEDIA_SOURCE_ID]/level" ,f (level in dB) ]
RX: [ "/media/source/[MEDIA_SOURCE_ID]/level" ,f (level in dB) ]
```

5.4.8 /media/source/[MEDIA_SOURCE_ID]/playstate

Optional read only values for a media source that has the ability to play media.:

```
TX: [ "/media/source/[MEDIA_SOURCE_ID]/playstate" ]
RX: [ "/media/source/[MEDIA_SOURCE_ID]/playstate" ,shd (transport state) (transport position in ms)
```

Transport state can be one of:

- "001 playing"
- "002 paused"
- "003 recording"

- "004 seeking"
- "005 scanning"

5.4.9 /media/source/[MEDIA_SOURCE_ID]/play

Optional command to start or query playback status. Changing play status may change the status of the other items such as "pause", "stop", "ff", "rw", "scanfwd" and "scanback."

Query the media source playback status:

```
TX: [ "/media/source/[MEDIA_SOURCE_ID]/play" ]
RX: [ "/media/source/[MEDIA_SOURCE_ID]/play" ,T ]
or:
RX: [ "/media/source/[MEDIA_SOURCE_ID]/play" ,F ]
```

Toggle the media source playback status:

```
TX: [ "/media/source/[MEDIA_SOURCE_ID]/play" ,I ]
RX: [ "/media/source/[MEDIA_SOURCE_ID]/play" ,T ]
or:
RX: [ "/media/source/[MEDIA_SOURCE_ID]/play" ,F ]
```

Set the media source to start playback:

```
TX: [ "/media/source/[MEDIA_SOURCE_ID]/play" ,T ]
RX: [ "/media/source/[MEDIA_SOURCE_ID]/play" ,T ]
```

Set the media source to stop playback:

```
TX: [ "/media/source/[MEDIA_SOURCE_ID]/play" ,F ]
RX: [ "/media/source/[MEDIA_SOURCE_ID]/play" ,F ]
```

5.4.10 /media/source/[MEDIA_SOURCE_ID]/pause

Optional command to start or query pause status. Changing pause status may change the status of the other items such as "play", "stop", "ff", "rw", "scanfwd" and "scanback."

Query the media source pause status:

```
TX: [ "/media/source/[MEDIA_SOURCE_ID]/pause" ]
RX: [ "/media/source/[MEDIA_SOURCE_ID]/pause" ,T ]
or:
RX: [ "/media/source/[MEDIA_SOURCE_ID]/pause" ,F ]
```

Toggle the media source pause mode:

```
TX: [ "/media/source/[MEDIA_SOURCE_ID]/pause" ,I ]
RX: [ "/media/source/[MEDIA_SOURCE_ID]/pause" ,T ]
or:
RX: [ "/media/source/[MEDIA_SOURCE_ID]/pause" ,F ]
```

Set the media source to pause:

```
TX: [ "/media/source/[MEDIA_SOURCE_ID]/pause" ,T ]
RX: [ "/media/source/[MEDIA_SOURCE_ID]/pause" ,T ]
```

Set the media source to unpause:

```
TX: [ "/media/source/[MEDIA_SOURCE_ID]/pause" ,F ]
RX: [ "/media/source/[MEDIA_SOURCE_ID]/pause" ,F ]
```

5.4.11 /media/source/[MEDIA_SOURCE_ID]/stop

Optional command to stop media playback or query stop status. Changing stop status may change the status of the other items such as “play”, “pause”, “ff”, “rw”, “scanfwd” and “scanback.”

Query the media source stop status:

```
TX: [ "/media/source/[MEDIA_SOURCE_ID]/stop" ]
RX: [ "/media/source/[MEDIA_SOURCE_ID]/stop" ,T ]
or:
RX: [ "/media/source/[MEDIA_SOURCE_ID]/stop" ,F ]
```

Set the media source to stop:

```
TX: [ "/media/source/[MEDIA_SOURCE_ID]/stop" ,I ]
RX: [ "/media/source/[MEDIA_SOURCE_ID]/stop" ,T ]
```

5.4.12 /media/source/[MEDIA_SOURCE_ID]/tracks

Optional command to query the number of tracks available in the media source.

Query the media source track count:

```
TX: [ "/media/source/[MEDIA_SOURCE_ID]/tracks" ]
RX: [ "/media/source/[MEDIA_SOURCE_ID]/tracks" ,i (track count) ]
```

5.4.13 /media/source/[MEDIA_SOURCE_ID]/track

Optional command to select a media playback “track” by track number. Changing the current track may change the status of the other items such as “play”, “stop”, “pause”, “ff”, “rw”, “scanfwd” and “scanback.”

Query the media source current track:

```
TX: [ "/media/source/[MEDIA_SOURCE_ID]/track" ]
RX: [ "/media/source/[MEDIA_SOURCE_ID]/track" ,i (current track number) ]
```

Set the media source to select another track:

```
TX: [ "/media/source/[MEDIA_SOURCE_ID]/track" ,i (track number) ]
RX: [ "/media/source/[MEDIA_SOURCE_ID]/track" ,i (current track number) ]
```

5.4.14 /media/source/[MEDIA_SOURCE_ID]/position

Optional address used to query the current position of media playback or to jump to a specific position. Changing the current position may change the status of the other items such as “play”, “stop”, “pause”, “ff”, “rw”, “scanfwd” and “scanback.”

Query the media source current position in milliseconds:

```
TX: [ "/media/source/[MEDIA_SOURCE_ID]/position/ms" ]
RX: [ "/media/source/[MEDIA_SOURCE_ID]/position/ms" ,h (current media time in ms) ]
```

Set the media source position in milliseconds:

```
TX: [ "/media/source/[MEDIA_SOURCE_ID]/position/ms" ,h (media time in ms) ]
RX: [ "/media/source/[MEDIA_SOURCE_ID]/position/ms" ,h (current media time in ms) ]
```

Query the media source current position in samples:

```
TX: [ "/media/source/[MEDIA_SOURCE_ID]/position/sample" ]
RX: [ "/media/source/[MEDIA_SOURCE_ID]/position/sample" ,h (current media time in samples) ]
```

Set the media source position in in samples:

```
TX: [ "/media/source/[MEDIA_SOURCE_ID]/position/sample" ,h (media time in samples) ]
RX: [ "/media/source/[MEDIA_SOURCE_ID]/position/sample" ,h (current media time in samples) ]
```

5.4.15 /media/source/[MEDIA_SOURCE_ID]/ff

Optional command to set the media source to be in “fast forward” mode. “Fast forward” mode moves the position of the media forward at some speed factor without actually playing the media content. Changing the fast forward mode may change the status of the other items such as “play”, “stop”, “pause”, “rw”, “scanfwd” and “scanback.”

Query the media source fast forward mode:

```
TX: [ "/media/source/[MEDIA_SOURCE_ID]/ff" ]
RX: [ "/media/source/[MEDIA_SOURCE_ID]/ff" ,Ti (speed factor) ]
or:
RX: [ "/media/source/[MEDIA_SOURCE_ID]/ff" ,Fi (speed factor) ]
```

Set the media source to be in fast forward mode:

```
TX: [ "/media/source/[MEDIA_SOURCE_ID]/ff" ,Ti (speed factor) ]
RX: [ "/media/source/[MEDIA_SOURCE_ID]/ff" ,Ti (speed factor) ]
```

Set the media source to be in fast forward mode without setting the speed factor:

```
TX: [ "/media/source/[MEDIA_SOURCE_ID]/ff" ,T ]
RX: [ "/media/source/[MEDIA_SOURCE_ID]/ff" ,T ]
```

Set the media source to not be in fast forward mode:

```
TX: [ "/media/source/[MEDIA_SOURCE_ID]/ff" ,F ]
RX: [ "/media/source/[MEDIA_SOURCE_ID]/ff" ,F ]
```

Toggle the media source fast forward mode:

```
TX: [ "/media/source/[MEDIA_SOURCE_ID]/ff" ,I ]
RX: [ "/media/source/[MEDIA_SOURCE_ID]/ff" ,T ]
or:
RX: [ "/media/source/[MEDIA_SOURCE_ID]/ff" ,T ]
```

5.4.16 /media/source/[MEDIA_SOURCE_ID]/rw

Optional command to set the media source to be in "rewind" mode. "Rewind" mode moves the position of the media backward at some speed factor without actually playing the media content. Changing the rewind mode may change the status of the other items such as "play", "stop", "pause", "ff", "scanfwd" and "scanback."

Query the media source rewind mode:

```
TX: [ "/media/source/[MEDIA_SOURCE_ID]/rw" ]
RX: [ "/media/source/[MEDIA_SOURCE_ID]/rw" ,Ti (speed factor) ]
or:
RX: [ "/media/source/[MEDIA_SOURCE_ID]/rw" ,Fi (speed factor) ]
```

Set the media source to be in rewind mode:

```
TX: [ "/media/source/[MEDIA_SOURCE_ID]/rw" ,Ti (speed factor) ]
RX: [ "/media/source/[MEDIA_SOURCE_ID]/rw" ,Ti (speed factor) ]
```

Set the media source to be in rewind mode without setting the speed factor:

```
TX: [ "/media/source/[MEDIA_SOURCE_ID]/rw" ,T ]
RX: [ "/media/source/[MEDIA_SOURCE_ID]/rw" ,T ]
```

Set the media source to not be in rewind mode:

```
TX: [ "/media/source/[MEDIA_SOURCE_ID]/rw" ,F ]
RX: [ "/media/source/[MEDIA_SOURCE_ID]/rw" ,F ]
```

Toggle the media source rewind mode:

```
TX: [ "/media/source/[MEDIA_SOURCE_ID]/rw" ,I ]
RX: [ "/media/source/[MEDIA_SOURCE_ID]/rw" ,T ]
or:
RX: [ "/media/source/[MEDIA_SOURCE_ID]/rw" ,T ]
```

5.4.17 /media/source/[MEDIA_SOURCE_ID]/scanfwd

Optional command to set the media source to be in "scan forward" mode. "Scan forward" mode moves the position of the media forward at some speed factor while playing snippets of the media content so the user can hear or see the current position. Changing the scan forward mode may change the status of the other items such as "play", "stop", "pause", "ff", "rw" and "scanback."

Query the media source scan forward mode:

```
TX: [ "/media/source/[MEDIA_SOURCE_ID]/scanfwd" ]
RX: [ "/media/source/[MEDIA_SOURCE_ID]/scanfwd" ,Ti (speed factor) ]
or:
RX: [ "/media/source/[MEDIA_SOURCE_ID]/scanfwd" ,Fi (speed factor) ]
```

Set the media source to be in scan forward mode:

```
TX: [ "/media/source/[MEDIA_SOURCE_ID]/scanfwd" ,Ti (speed factor) ]
RX: [ "/media/source/[MEDIA_SOURCE_ID]/scanfwd" ,Ti (speed factor) ]
```

Set the media source to be in scan forward mode without setting the speed factor:

```
TX: [ "/media/source/[MEDIA_SOURCE_ID]/scanfwd" ,T ]
RX: [ "/media/source/[MEDIA_SOURCE_ID]/scanfwd" ,T ]
```

Set the media source to not be in scan forward mode:

```
TX: [ "/media/source/[MEDIA_SOURCE_ID]/scanfwd" ,F ]
RX: [ "/media/source/[MEDIA_SOURCE_ID]/scanfwd" ,F ]
```

Toggle the media source scan forward mode:

```
TX: [ "/media/source/[MEDIA_SOURCE_ID]/scanfwd" ,I ]
RX: [ "/media/source/[MEDIA_SOURCE_ID]/scanfwd" ,T ]
or:
RX: [ "/media/source/[MEDIA_SOURCE_ID]/scanfwd" ,T ]
```

5.4.18 /media/source/[MEDIA_SOURCE_ID]/scanback

Optional command to set the media source to be in "scan backward" mode. "Scan backward" mode moves the position of the media backward at some speed factor while playing snippets of the media content so the user can hear or see the current position. Changing the scan backward mode may change the status of the other items such as "play", "stop", "pause", "ff", "rw" and "scanfwd."

Query the media source scan backward mode:

```
TX: [ "/media/source/[MEDIA_SOURCE_ID]/scanback" ]
RX: [ "/media/source/[MEDIA_SOURCE_ID]/scanback" ,Ti (speed factor) ]
or:
RX: [ "/media/source/[MEDIA_SOURCE_ID]/scanback" ,Fi (speed factor) ]
```

Set the media source to be in scan backward mode:

```
TX: [ "/media/source/[MEDIA_SOURCE_ID]/scanback" ,Ti (speed factor) ]
RX: [ "/media/source/[MEDIA_SOURCE_ID]/scanback" ,Ti (speed factor) ]
```

Set the media source to be in scan backward mode without setting the speed factor:

```
TX: [ "/media/source/[MEDIA_SOURCE_ID]/scanback" ,T ]
RX: [ "/media/source/[MEDIA_SOURCE_ID]/scanback" ,T ]
```

Set the media source to not be in scan backward mode:

```
TX: [ "/media/source/[MEDIA_SOURCE_ID]/scanback" ,F ]
RX: [ "/media/source/[MEDIA_SOURCE_ID]/scanback" ,F ]
```

Toggle the media source scan backward mode:

```
TX: [ "/media/source/[MEDIA_SOURCE_ID]/scanback" ,I ]
RX: [ "/media/source/[MEDIA_SOURCE_ID]/scanback" ,T ]
or:
RX: [ "/media/source/[MEDIA_SOURCE_ID]/scanback" ,T ]
```

5.5 Media Sink Control - `/media/sink/`

A "Media Sink" is defined as a destination of single channel or a logical collection of channels of audio or a combined audio/video stream receiver.

Examples of "Media Sinks" may be:

- A speaker or analog line output
- The left and right channels of a SPDIF or AES3 digital audio output
- A single audio record channel to a hard disk recorder system
- A number of related audio record channels to a hard disk recorder system
- A combined audio and video stream such as MPEG4 which may have multiple audio channels as well as a video channel.

In the OSC addresses, `[MEDIA_SINK_ID]` used here represents some unique identifier for a media sink on this device.

The following addresses are required to be available:

- `/media/sink/[MEDIA_SINK_ID]/id`
- `/media/sink/[MEDIA_SINK_ID]/type`
- `/media/sink/[MEDIA_SINK_ID]/description`
- `/media/sink/[MEDIA_SINK_ID]/channels`

The following addresses are defined but are optional depending on the ability or feature set of the device:

- `/media/sink/[MEDIA_SINK_ID]/mute`
- `/media/sink/[MEDIA_SINK_ID]/level`
- `/media/sink/[MEDIA_SINK_ID]/pan`

The implementor can add any other addresses as necessary for the device. Examples of additional addresses could items such as:

- `/media/sink/[MEDIA_SINK_ID]/vendor/[VENDOR_OUI]/meter`
- `/media/sink/[MEDIA_SINK_ID]/vendor/[VENDOR_OUI]/scale`
- `/media/sink/[MEDIA_SINK_ID]/vendor/[VENDOR_OUI]/tone`
- `/media/sink/[MEDIA_SINK_ID]/vendor/[VENDOR_OUI]/xtrabass`
- `/media/sink/[MEDIA_SINK_ID]/vendor/[VENDOR_OUI]/record`

These additional addresses would not need to fit any specific form or function. Since they are discoverable via the `/osc/schema` and `/osc/limits` addresses any client can know how to represent these additional addresses to the end user.

5.5.1 /media/sink/[MEDIA_SINK_ID]/id

Read only value.

Get the current identifier for the specified media sink:

```
TX: [ "/media/sink/[MEDIA_SINK_ID]/id" ]
RX: [ "/media/sink/[MEDIA_SINK_ID]/id" ,i 1 ]
```

5.5.2 /media/sink/[MEDIA_SINK_ID]/type

Read only value.

Returns the user readable ASCII type of the specified media sink.

Reading:

```
TX: [ "/media/sink/[MEDIA_SINK_ID]/type" ]
RX: [ "/media/sink/[MEDIA_SINK_ID]/type" ,s (media type string) ]
```

Examples of media type strings could be:

- “Analog Microphone Audio Output”
- “Analog Line Out Audio Output”
- “Speaker Output”
- “AES/EBU Digital Output L”
- “AES/EBU Digital Output R”

note: This may need to be enumerated.

5.5.3 /media/sink/[MEDIA_SINK_ID]/description

Read only value.

Returns the user readable ASCII description of the specified media sink.

Reading:

```
TX: [ "/media/sink/[MEDIA_SINK_ID]/description" ]
RX: [ "/media/sink/[MEDIA_SINK_ID]/description" ,s (media description string) ]
```

Examples of media type strings could be:

- “Front Panel Line Out #1”
- “Back Panel Speaker Output Right Channel”
- “Hard Disk Record Channel #5”
- “MIDI Output Stream”
- “SMPTE LTC Output”

5.5.4 /media/sink/[MEDIA_SINK_ID]/channels

Read only value.

Returns the count of media channels for this media sink.

Reading:

```
TX: [ "/media/sink/[MEDIA_SINK_ID]/channels" ]
RX: [ "/media/sink/[MEDIA_SINK_ID]/channels" ,i (media sink channel count) ]
```

5.5.5 /media/sink/[MEDIA_SINK_ID]/mute

Optional read/write boolean value.

A typetag of 'T' means the channel is muted. A typetag of 'F' means the channel is not muted.

Get the current mute setting for the specified media sink:

```
TX: [ "/media/sink/[MEDIA_SINK_ID]/mute" ]
RX: [ "/media/sink/[MEDIA_SINK_ID]/mute" ,T ]
or:
RX: [ "/media/sink/[MEDIA_SINK_ID]/mute" ,F ]
```

Set the media source to be muted:

```
TX: [ "/media/sink/[MEDIA_SINK_ID]/mute" ,T ]
RX: [ "/media/sink/[MEDIA_SINK_ID]/mute" ,T ]
```

Set the media source to be unmuted:

```
TX: [ "/media/sink/[MEDIA_SINK_ID]/mute" ,F ]
RX: [ "/media/sink/[MEDIA_SINK_ID]/mute" ,F ]
```

5.5.6 /media/sink/[MEDIA_SINK_ID]/pan

Optional read/write float32 value for media panpot from -1.0 to +1.0

Get the current pan setting for the specified media sink:

```
TX: [ "/media/sink/[MEDIA_SINK_ID]/pan" ]
RX: [ "/media/sink/[MEDIA_SINK_ID]/pan" ,f (value) ]
```

Set the current pan setting:

```
TX: [ "/media/sink/[MEDIA_SINK_ID]/pan" ,f (value) ]
RX: [ "/media/sink/[MEDIA_SINK_ID]/pan" ,f (value) ]
```

For media sinks that have more than two channels, the pan setting may respond to multiple values/dimensions.

5.5.7 /media/sink/[MEDIA_SINK_ID]/level

Optional read/write float32 value for media sink level in dB.

Get the current level setting for the specified media sink:

```
TX: [ "/media/sink/[MEDIA_SINK_ID]/level" ]
RX: [ "/media/sink/[MEDIA_SINK_ID]/level" ,f (level in dB) ]
```

Set the media source level:

```
TX: [ "/media/sink/[MEDIA_SINK_ID]/level" ,f (level in dB) ]
RX: [ "/media/sink/[MEDIA_SINK_ID]/level" ,f (level in dB) ]
```

5.6 Global Schema

The global schema prefixes allow a simple client to broadcast or multicast a message to all devices on a network or in a group and have only the specified device, or members of the specified system, or devices produced by a specific vendor respond.

5.6.1 /bydevice/[DEVICE_NAME]/

Alias to / on all devices matching device name [DEVICE_NAME]

5.6.2 /bysystem/[SYSTEM_NAME]/

Alias to / on all devices matching system name [SYSTEM_NAME]

5.6.3 /byvendor/[VENDOR_OUI]/

Alias to / on all devices with the specified vendor oui.

5.7 Vendor Specific Device Settings

5.7.1 /vendor/[VENDOR_OUI]/

Any device specific vendor specific entries for any purpose may be added to the /vendor/[VENDOR_OUI]/ address container.

CHAPTER

SIX

EXAMPLE AVBC TRANSACTIONS

6.1 Subscribe to all devices stream source and sink state

Broadcast/multicast request to all devices on the network to subscribe to all active stream source and sink states:

```
TX: [ "/osc/state/subscribe" ,s "/bysystem/*/bydevice/*/avb/*/*/state" ]
```

After this message is accepted, every device on the network will transmit their results to the client via unicast UDP. Whenever the state of an avb stream source or an avb stream sink changes, the client will be notified.

6.2 `/osc/limits` Example

For the optional address for media sink 1's level:

```
TX: [ "/media/sink/1/level" ]
RX: [ "/media/sink/1/level" ,f (level in dB) ]
```

To request the limits for "/media/sink/1/level":

```
TX: [ "/osc/limits/media/sink/1/level" ]
RX: [ "/osc/limits/media/sink/1/level" ,[sssfsfsfss]
      "type" "f" "min" -100.0 "max" 10.0
      "inc" 0.1 "units" "dB" ]
```

6.3 Create AVB Stream Source

A talker device has some AES3 digital audio inputs designated as stereo media sources #1,#2 and #3. The client wants to take the audio from these AES3 channels and create a single 6 channel audio stream suitable for a 5.1 audio format with the default presentation time.:

```
TX: [ "/avb/source/create" ,NNssNiiiiiii "Movie Audio" "(Format codes TBD)"
      6 0x00010000 0x00010001 0x00020000 0x00020001 0x00030000 0x0003001 ]
RX: [ "/avb/source/create" ,hhsshiiiiiii (stream id) (multicast MAC address)
      "Movie Audio" "(Format codes TBD)" (presentation time offset in ns)
      6 0x00010000 0x00010001 0x00020000 0x00020001 0x00030000 0x0003001 ]
```

6.4 Create AVB Stream Sink based on Stream ID

Create an AVB stream sink by specifying a stream id:

```
TX: [ "/avb/sink/create" ,NNNhNhii... (stream id)
      (presentation) (number of channels) (media sink channel code)... ]
RX: [ "/avb/sink/create" ,NNhhhhii... (sink id) (stream id) (multicast mac address)
      (presentation) (number of channels) (media sink channel code)... ]
```

6.5 Create AVB Stream Sink based on Stream ID and Multicast Mac Address

Multicast MAC address is required to be pre-known to do SRP for streams that have been pruned by talker-pruning.

Create an AVB stream sink by specifying a stream id and multicast mac address:

```
TX: [ "/avb/sink/create" ,NNNhhhii... (stream id) (multicast mac address)
      (presentation) (number of channels) (media sink channel code)... ]
RX: [ "/avb/sink/create" ,NNhhhhii...(sink id) (stream id) (multicast mac address)
      (presentation) (number of channels) (media sink channel code)... ]
```

6.6 Create AVB Stream Sink by Device Name and Stream Name

Create an AVB stream sink by specifying a device name and stream name:

```
TX: [ "/avb/sink/create" ,ssNNhii... (device name) (stream name) (presentation)
      (number of channels) (channel map)... ]
RX: [ "/avb/sink/create" ,sshhhhii... (device name) (stream name) (sink id) (stream id)
      (multicast mac address) (presentation) (number of channels) (channel map)... ]
```

CHAPTER

SEVEN

APPENDICES

7.1 Complete Minimum Required Schema

```
"/avb/"
"/avb/sink/"
"/avb/sink/[SINK_ID]/channels"
"/avb/sink/[SINK_ID]/destroy"
"/avb/sink/[SINK_ID]/format"
"/avb/sink/[SINK_ID]/id"
"/avb/sink/[SINK_ID]/map"
"/avb/sink/[SINK_ID]/presentation"
"/avb/sink/[SINK_ID]/source/"
"/avb/sink/[SINK_ID]/source/device"
"/avb/sink/[SINK_ID]/source/id"
"/avb/sink/[SINK_ID]/source/mmac"
"/avb/sink/[SINK_ID]/source/name"
"/avb/sink/[SINK_ID]/state"
"/avb/sink/create"
"/avb/sink/formats"
"/avb/source/"
"/avb/source/[STREAM_ID]/"
"/avb/source/[STREAM_ID]/channels"
"/avb/source/[STREAM_ID]/destroy"
"/avb/source/[STREAM_ID]/format"
"/avb/source/[STREAM_ID]/id"
"/avb/source/[STREAM_ID]/map"
"/avb/source/[STREAM_ID]/mmac"
"/avb/source/[STREAM_ID]/presentation"
"/avb/source/[STREAM_ID]/state"
"/avb/source/byname/"
"/avb/source/byname/[STREAM_NAME]/"
"/avb/source/create"
"/avb/source/formats"
"/bydevice/"
"/bydevice/[DEVICE_NAME]/"
"/bysystem/"
"/bysystem/[SYSTEM_NAME]/"
"/byvendor/"
"/byvendor/[VENDOR_OUI]/"
"/device/"
"/device/identity/product"
"/device/identity/serial"
"/device/identity/vendor"
```

```
"/device/identity/vendor_id"
"/device/identity/version"
"/device/name"
"/device/system"
"/media/"
"/media/sink/"
"/media/sink/[MEDIA_SINK_ID]/"
"/media/sink/[MEDIA_SINK_ID]/channels"
"/media/sink/[MEDIA_SINK_ID]/description"
"/media/sink/[MEDIA_SINK_ID]/id"
"/media/sink/[MEDIA_SINK_ID]/type"
"/media/source/"
"/media/source/[MEDIA_SOURCE_ID]/"
"/media/source/[MEDIA_SOURCE_ID]/channels"
"/media/source/[MEDIA_SOURCE_ID]/description"
"/media/source/[MEDIA_SOURCE_ID]/id"
"/media/source/[MEDIA_SOURCE_ID]/type"
"/osc/"
"/osc/limits/"
"/osc/schema/"
"/osc/subscribe"
"/osc/type/"
"/osc/type/accepts"
"/osc/type/reports"
"/osc/version"
```

7.2 Complete Optional Schema

```
"/avb/sink/[SINK_ID]/vendor/"
"/avb/sink/[SINK_ID]/vendor/[VENDOR_OUI]/*"
"/avb/sink/vendor/"
"/avb/sink/vendor/[VENDOR_OUI]/*"
"/avb/source/[STREAM_ID]/vendor/"
"/avb/source/[STREAM_ID]/vendor/[VENDOR_OUI]/*"
"/avb/source/vendor/"
"/avb/source/vendor/[VENDOR_OUI]/*"
"/avb/vendor/"
"/avb/vendor/[VENDOR_OUI]/*"
"/device/identity/vendor/"
"/device/identity/vendor/[VENDOR_OUI]/*"
"/device/vendor/"
"/device/vendor/[VENDOR_OUI]/*"
"/media/sink/[MEDIA_SINK_ID]/level"
"/media/sink/[MEDIA_SINK_ID]/mute"
"/media/sink/[MEDIA_SINK_ID]/pan"
"/media/sink/[MEDIA_SINK_ID]/vendor/"
"/media/sink/[MEDIA_SINK_ID]/vendor/[VENDOR_OUI]/*"
"/media/sink/vendor/"
"/media/sink/vendor/[VENDOR_OUI]/*"
"/media/source/[MEDIA_SOURCE_ID]/ff"
"/media/source/[MEDIA_SOURCE_ID]/level"
"/media/source/[MEDIA_SOURCE_ID]/mute"
"/media/source/[MEDIA_SOURCE_ID]/pan"
"/media/source/[MEDIA_SOURCE_ID]/pause"
"/media/source/[MEDIA_SOURCE_ID]/play"
```

```
"/media/source/[MEDIA_SOURCE_ID]/playstate"
"/media/source/[MEDIA_SOURCE_ID]/position"
"/media/source/[MEDIA_SOURCE_ID]/rew"
"/media/source/[MEDIA_SOURCE_ID]/stop"
"/media/source/[MEDIA_SOURCE_ID]/track"
"/media/source/[MEDIA_SOURCE_ID]/vendor/"
"/media/source/[MEDIA_SOURCE_ID]/vendor/[VENDOR_OUI]/"
"/media/source/vendor/"
"/media/source/vendor/[VENDOR_OUI]/*"
"/media/vendor/"
"/media/vendor/[VENDOR_OUI]/*"
"/vendor/"
"/vendor/[VENDOR_OUI]/*"
```

7.3 Definitions

Active Stream A successful Qat reservation associated with a given Talker and data is flowing

Channel One component of stream (i.e., the left channel of a stereo stream)

JSON Javascript Object Notation

Listener An AVB end node that sinks streams

Potential Stream A stream that is advertised but has no Listeners associated with it

Reserved Stream A successful Qat reservation associated with a given Talker but data is not flowing

Stream Sink Destination of a single 1722 stream

Stream Source Source of a single 1722 stream

Talker An AVB end node that sources streams

7.4 Reference RFC's and standards

OSC v1.0: http://www.opensoundcontrol.org/spec-1_0

OSC v1.1: http://www.opensoundcontrol.org/spec-1_1

Center For New Music and Audio Technology: http://cnmat.berkeley.edu/

oscpack: http://www.audiomulch.com/~rossb/code/oscpack/

micro-OSC: http://cnmat.berkeley.edu/research/uosc

HTTP RFC 2616: http://tools.ietf.org/html/rfc2616

JSON RFC 4627: http://tools.ietf.org/html/rfc4627

7.5 HTTP to OSC Bridge

For devices that may want to include a HTTP server that delivers a rich web application for the user to manage the device, it is recommended that the the device contain a HTTP/JSON to OSC bridge as defined here.

7.5.1 JSON Representation of OSC messages

JavaScript Object Notation (JSON) is a common, efficient method to transfer data structures over HTTP to or from a javascript web application running in a web browser.

JSON is defined in rfc4627: http://www.ietf.org/rfc/rfc4627.txt

Various forms of OSC messages or bundles can easily be represented by JSON.

An OSC message encoded as JSON always has an "a" entry with a string value for the OSC address, and a "t" entry with a string value of the typetags. If the typetags specified require actual values to be encoded, the values are encoded in an "v" entry which is always an array of the values.

The values in the "v" array will always be javascript fundamental types such as strings or integers or double precision floating point or will be a structure type as defined in this table:

Type Tag	Type of corresponding argument
c	a javascript string with one character
d	a javascript floating point number
h	a javascript string containing the hex representation of the 64 bit integer
r	a javascript array of numbers: [R, G, B, A]
S	a javascript string
m	4 byte MIDI message as array of numbers: [portid, status, data1, data2]
[	A javascript array begins
]	A javascript array ends
N	nothing
T	nothing
F	nothing
R	nothing
E	a javascript string

OSC Message

OSC Message:

```
[ "/ADDRESS" TYPETAGS VALUES...]
```

JSON Representation:

```
{
   "a" : "/ADDRESS",
   "t" : "TYPETAGS",
   "v" : [ VALUE0, VALUE1, VALUE2 ... ]
}
```

Example OSC Message:

```
[ "/osc/limits" ,s "/media/in/1/gain" ]
```

JSON Representation:

```
{
   "a" : "/osc/limits",
   "t" : "s",
   "v" : [ "/media/in/1/gain" ]
}
```

OSC Bundle

In an HTTP transfer, multiple messages may be grouped together in a javascript array and they would be handled together.

However, an OSC Bundle is a different form, and contains a time field as well.

The JSON representation of an OSC Bundle always contains the entries "time_s" (seconds part of time), "time_ns" (nanosecond part of time) and "msgs" (array of messages).

Example OSC Bundle:

```
Time: 456.123 seconds
[ "/media/in/1/mute" ,T ]
[ "/media/in/2/mute" ,T ]
```

JSON Representation:

```
{
   "time_s" : 456,
   "time_ns" : 123000000,
   "msgs"  : [
                { "a" : "/media/in/1/mute", "t" : "T" },
                { "a" : "/media/in/2/mute", "t" : "T" }
                ]
}
```

7.5.2 HTTP PUT REQUEST

All OSC messages that are encoded as JSON and sent to the HTTP-OSC bridge are sent via the HTTP "PUT" method. The response of the HTTP request is all of the JSON encoded OSC messages that are in response to the original request.

Example 1

HTTP Client sets input 1 scale to 1:

```
PUT /osc/ HTTP/1.1
Content-Type: application/json
Connection: close

{
   "a" : "/media/in/1/scale",
   "t" : "i",
   "v" : [ 1 ]
}
```

AVBC HTTP Server Replies:

```
HTTP/1.1 200 OK
Content-Type: application/json
Connection: close

{
  "a" : "/media/in/1/scale",
```

```
  "t" : "i",
  "v" : [1]
}
```

Example 2

HTTP Client sets all inputs to mute:

```
PUT /osc/ HTTP/1.1
Content-Type: application/json
Connection: close

{
   "a" : "/media/in/*/mute",
   "t" : "T"
}
```

AVBC HTTP Server replies with multiple message responses for all affected addresses (assume a device with 4 media inputs) in a javascript array:

```
HTTP/1.1 200 OK
Content-Type: application/json
Connection: close

[
      {
         "a" : "/media/in/1/mute",
         "t" : "T"
      },
      {
         "a" : "/media/in/2/mute",
         "t" : "T"
      },
      {
         "a" : "/media/in/3/mute",
         "t" : "T"
      },
      {
         "a" : "/media/in/4/mute",
         "t" : "T"
      }
]
```

INDEX