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

APRIL 7, 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, Open Show Control, Digital Audio, Digital Video, Schema.

Jeff Koftinoff <jeff.koftinoff@gmail.com>

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

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 prior knowledge of.

As it is also understood that the HTTP Web Browser is the ubuquitous client application, this specification also defines a 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/

It uses a support library called microsupport which is also open source BSD licensed and will be available at:

http://microsupport.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.

CHAPTER

FIVE

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.

5.1 Standard OSC Syntax

5.1.1 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).

5.1.2 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.

5.1.3 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.

5.2 OSC Arguments

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

5.3 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	

5.4 OSC Semantics

5.4.1 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.)

1. 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.
2. The "//" pattern matches any strings between '/' characters, including other '/' characters.
3. Any other character in an OSC Address Pattern can match only the same character.

5.5 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.

CHAPTER

SIX

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.

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 IPv6 with fallback to IPv4 when necessary.
- MAY support TCP streams for OSC Message/Bundle transport.

A device functioning as an “AVBC Protocol Server”:

- 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.
- MUST be able to join an IPv4 Multicast Address via IGMP in order to transmit state changes to interested parties
- MAY support IPv6 with fallback to IPv4 when necessary.
- MAY support TCP streams for OSC Message/Bundle transport.

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

When a device 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 - for instance different programs on a desktop computer - and each client can send a multicast UDP packet to the network. When an AVB Protocol Server recieves the message, it MUST respond to the message via a unicast message addressed to the source IP address and port from the request.

Clients always initiate requests to servers. A client can:

- Send a request to a server via a unicast UDP message
- Send a request to a server via a multicast UDP message
- Send a request to a server via a TCP stream (if both devices support TCP streams)

CHAPTER

SEVEN

HTTP TO OSC BRIDGE

7.1 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 integer
r	a javascript array of integers: [R, G, B, A]
S	a javascript string
m	4 byte MIDI message as array of integers: [portid, status, data1, data2]
[	A javascript array begins
]	A javascript array ends

7.1.1 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" ]
}
```

7.1.2 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” and “msgs”.

Example OSC Bundle:

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

JSON Representation:

```
{
   "time" : 456.123,
   "msgs"  : [
                { "a" : "/media/in/1/mute", "t" : "T" },
                { "a" : "/media/in/2/mute", "t" : "T" }
                ]
}
```

7.2 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.

7.2.1 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]
}
```

7.2.2 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"
     }
]
```

CHAPTER

EIGHT

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:

8.1 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

8.1.1 `/osc/version`

Read only value.

Report the OSC Version implemented in the server:

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

8.1.2 `/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 "ifsbhTF" ]

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

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*

8.1.3 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" ,s "/avb/" ]
```

Would return the following OSC Message, describing the addresses that are contained in the "/avb/" one level deeper:

```
RX: [ "/osc/schema" ,sssssss "/avb/" "sources" "source/"
      "sinks/" "sink/" "loops/" "loop/" ]
```

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" ,s "/avb/source/" ]
```

And it would return the following OSC Message:

```
RX: [ "/osc/schema" ,ssssssss "/avb/source/" "sync" "formats"
      "format" "channels" "name" "id" "presentation" ]
```

In the case of an invalid request, the result would be of type 'N', for nil:

```
TX: [ "/osc/schema" ,s "/some/nonexistant/address" ]
RX: [ "/osc/schema" ,sN "/some/nonexistant/address" ]
```

8.1.4 The Limits Predicate - /osc/limits

The /osc/limits predicate can be passed to any other OSC request. The response of the request is always four parameters describing value limits of the addressed object:

- The typetags used for the object
- The writability of the address, either "rw" or "ro"
- The minimum value of the object
- The maximum value of the object
- The recommended step size of the object
- The object's unit type in UTF-8 if it has units; for example:: mm, dB, dBu, dBfs, Hz

Examples of `/osc/limits` usage

For the fictituous address for media input 1's gain:

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

To request the limits:

```
TX: [ "/osc/limits" ,s "/media/in/1/gain" ]
RX: [ "/osc/limits" ,sssfffs "/media/in/1/gain"
      "fi" "rw" -100.0 10.0 0.1 "dB" ]
```

The response of the limits request shows that this address/parameter combination accepts values in both `float32` and `int32` formats. 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.

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.

8.2 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.

8.2.1 `/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) ]
```

8.2.2 `/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) ]
```

8.2.3 /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) ]
```

8.2.4 /device/identity/version

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

- A *string* containing the product's firmware/software version

Getting device version:

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

8.2.5 /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

Getting device product serial number:

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

8.2.6 /device/name

The `/device/name` address is used as an end-user configurable name for the device.

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) ]
```

8.2.7 /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) ]
```

8.3 AVB Control - /avb/

The `/avb/` address holds addresses that are used for configuration of all the AVB stream handlers of all types in the module.

These include AVB Stream Sources, and AVB Stream Sinks.

A device can have none, one, or many stream handlers for each of these.

Each handler will have a name attributed to it.

The default name for a stream handler is the concatenation of the following fields:

- Device vendor_id
- Device model_id
- Device serial number
- Stream handler type (“source”, “sink”)
- Stream handler number

Each field is separated by dashes.

For example, the first source stream handler in a device from vendor “123456789abc” with model number 987654321 and serial number WK9H42XA3 and might be named:

```
"123456789a-987654321-WK9H42XA3-source-1"
```

In a message querying or setting parameters for these stream handlers, the stream handler can be referred to by number when the first parameter is an `i` type, or can be referred to by name, when the first parameter is an `s` type.

8.3.1 AVB Stream Sources

/avb/sources

Read only value.

Get count of how many stream sources the device can provide at the same time:

```
TX: [ "/avb/sources" ]
RX: [ "/avb/sources" ,i (value) ]
```

/avb/source/byname/NAME

Alias to the associated `/avb/source/#/` that matches the `NAME`

/avb/source/byid/ID

Alias to the associated `/avb/source/#/` that matches the `ID`

/avb/source/#/id

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

Request current stream id:

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

/avb/source/#/name

This read/write value specifies the UTF-8 name for the stream source. This UTF-8 name must be unique within the same system.

Request current stream name:

```
TX: [ "/avb/source/#/name" ]
RX: [ "/avb/source/#/name" ,s (name) ]
```

Set the stream name:

```
TX: [ "/avb/source/#/name" ,s (new name) ]
RX: [ "/avb/source/#/name" ,s (new name) ]
```

/avb/source/#/state

Read only value

Get current state of this stream source:

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

Valid stream states are:

- “potential”
- “reserved”
- “active”

/avb/source/#/sync/#

Read/write value.

Get the current synchronization sources for this talker stream:

```
TX: [ "/avb/source/#/sync/#" ]
RX: [ "/avb/source/#/sync/#" ,ii (avb sink #) (priority) ]
```

Set the current synchronization source for this talker stream:

```
TX: [ "/avb/source/#/sync/#" ,ii (avb sink #) (priority) ]
RX: [ "/avb/source/#/sync/#" ,ii (avb sink #) (priority) ]
```

/avb/source/#/formats

Read only value.

Get a list of all the 1722 formats that are supported by the specified talker stream:

```
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/#/format

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

Request current format:

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

Set the transmission format:

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

Request current channel count for stream:

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

Set channel count for stream:

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

/avb/source/#/map

Request the current channel map for the Stream Sink:

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

Set expected channel map for stream:

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

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

Set the talker's presentation time offset:

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

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

8.3.2 AVB Stream Sinks

/avb/sinks

Read only value.

Get count of how many streams the device can receive at the same time:

```
TX: [ "/avb/sinks" ]
RX: [ "/avb/sinks" ,i (value) ]
```

/avb/sink/byname/NAME

Alias to the associated `/avb/sink/#/` that matches the `NAME`

/avb/sink/byid/ID

Alias to the associated `/avb/sink/#/` that matches the `ID`

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

/avb/sink/#/name

The name of the Stream Sink handler.

Request current Stream Sink handler name:

```
TX: [ "/avb/sink/#/name" ]
RX: [ "/avb/sink/#/name" ,s (handler name) ]
```

set the current Stream Sink handler name:

```
TX: [ "/avb/sink/#/name" ,s (handler name) ]
RX: [ "/avb/sink/#/name" ,s (handler name) ]
```

/avb/sink/#/formats

Read only value.

Get a list of all the 1722 formats that are supported by the specified stream sink:

```
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/#/format

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

Request current format:

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

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

/avb/sink/#/channels

Request current expected channel count for stream:

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

Set expected channel count for stream:

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

/avb/sink/#/map

Request the current channel map for the Stream Sink:

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

Set expected channel map for stream:

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

/avb/sink/#/source

This read/write value specifies the UTF-8 name for the system and the stream that the listener is currently subscribed to.

Request current subscription stream system and name:

```
TX: [ "/avb/sink/#/source" ]
RX: [ "/avb/sink/#/source" ,ss (system name) (stream name) ]
```

Set the listener to subscribe to the specified system and stream name:

```
TX: [ "/avb/sink/#/source" ,ss (system name) (stream name) ]
RX: [ "/avb/sink/#/source" ,ss (system name) (stream name) ]
```

If the **system name** is set to a blank string, the system name will default to the system name that the device is currently within,

/avb/sink/#/id

This read/write value specifies the full stream ID for that this listener is subscribed to. 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.

Normally, the subscribed stream id is determined by the listener device looking up the sink source's stream id by system name and stream name. However the client can override the lookup by name and set a full stream id to subscribe to directly.

Request current stream id:

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

Set the subscribed stream id:

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

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

Set the sink's presentation time offset:

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

All stream presentation time offset values default to 0 nanoseconds.

8.4 Media Input Control - /media/in/

A "Media Input" is defined as a single channel of audio or a combined audio/video stream going in to the AVB endpoint via non-avb methods.

The term "Input" implies that the media is coming from an external source such as a microphone or a camera - but it doesn't have to be an external signal, for it could be an signal that is internally generated by the device.

An example of an internally generated signal would be the playback channels from a hard disk playback system. These signals would be presented as "Inputs" to the AVB endpoint and then could be presented to the AVB network as AVB Stream Sources.

In these OSC addresses, the "#" character used here represents the ascii representation of the media input number, starting at 1.

The following addresses are required to be available:

- /media/ins
- /media/in/byname/NAME
- /media/in/#/name
- /media/in/#/type

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

- /media/in/#/mute
- /media/in/#/level
- /media/in/#/meter
- /media/in/#/phantom
- /media/in/#/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.

8.4.1 /media/ins

Read only value.

Specifies number of media inputs on the device which can map to stream sources.

Reading:

```
TX: [ "/media/ins" ]
RX: [ "/media/ins" ,i (count of available media inputs)
```

8.4.2 /media/in/byname/NAME

Alias to the associated `/media/in/#/` media input that has the specified `NAME`.

8.4.3 /media/in/#/name

Read/Write value.

Get the current name for the specified media input:

```
TX: [ "/media/in/#/name" ]
RX: [ "/media/in/#/name" ,s "The input name" ]
```

Set the name for the specified media input:

```
TX: [ "/media/in/#/name" ,s "New input name" ]
RX: [ "/media/in/#/name" ,s "New input name" ]
```

8.4.4 /media/in/#/type

Read only value.

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

Reading:

```
TX: [ "/media/in/#/type" ]
RX: [ "/media/in/#/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”

8.5 Media Output Control - /media/out/

A "Media Output" is defined as a single channel of audio or a combined audio/video stream coming out of the AVB endpoint via non-avb methods.

The term "Output" imples that the media is going to an external connection but it doesn't have to be - For instance it could be a channel that is going to an internal hard disk for recording.

In the OSC addresses, the "#" character used here represents the ascii representation of the media output number, starting at 1.

The following addresses are required to be available:

- /media/outs
- /media/out/byname/NAME
- /media/out/#/name
- /media/out/#/type

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

- /media/out/#/mute
- /media/out/#/level
- /media/out/#/meter
- /media/out/#/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.

8.5.1 /media/outs

Read only value.

Specifies number of media outputs on the device which can map to stream sources.

Reading:

```
TX: [ "/media/outs" ]
RX: [ "/media/outs" ,i (count of available media inputs)
```

8.5.2 /media/out/byname/NAME

Alias to the associated /media/out/#/ media input that has the specified NAME.

8.5.3 /media/out/#/name

Read/Write value.

Get the current name for the specified media out:

```
TX: [ "/media/out/#/name" ]
RX: [ "/media/out/#/name" ,s "The output name" ]
```

Set the name for the specified media input:

```
TX: [ "/media/out/#/name" ,s "New output name" ]
RX: [ "/media/out/#/name" ,s "New output name" ]
```

8.5.4 /media/out/#/type

Read only value.

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

Reading:

```
TX: [ "/media/out/#/type" ]
RX: [ "/media/out/#/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”

CHAPTER

NINE

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

CHAPTER

TEN

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

INDEX