AUDIX®
MICROPHONES

NDC Protocol & API v1.0.1

Audix Corporation Inc.
https://audixusa.com

January 20, 2020

Document Version: v1.2.2

Contents

1 Overview **1**

2 Conventions **1**

3 Compatibility **1**

4 Server Discovery **2**
4.1 Multicast DNS (mDNS) . . . 2

5 JSON-RPC Message Rules **2**
5.1 JSON-RPC Object Formatting . . . 3
5.2 JSON-RPC Object Usage . . . 3
5.3 Error Responses . . . 4

6 NDC Description **5**
6.1 Acquire Control of NDC Server . . . 6
6.2 Enumeration . . . 6

7 NDC Methods **7**
7.1 acquire_control . . . 9
7.2 release_control . . . 9
7.3 get_device_name . . . 10
7.4 get_device_firmware_ver . . . 10
7.5 get_device_protocol_ver . . . 11
7.6 get_jack_cnt . . . 11
7.7 get_jack_attr . . . 12
7.8 get_mic_id . . . 13
7.9 get_btn . . . 13
7.10 get_led . . . 14
7.11 set_led . . . 14
7.12 get_gain_vals . . . 15
7.13 get_gain . . . 15
7.14 set_gain . . . 16
7.15 get_hpf_vals . . . 16
7.16 get_hpf . . . 17
7.17 set_hpf . . . 17
7.18 get_lpf_vals . . . 19
7.19 get_lpf . . . 19
7.20 set_lpf . . . 20

Appendices **21**

A Mic ID Values **21**

References **22**

List of Figures

6.0.1 Common NDC client startup example 5
6.1.1 Acquire lock prevents secondary client communication with an acquired server 6

List of Tables

4.0.1 NDC Server Default Socket Configuration 2
5.3.1 Error Codes in JSON-RPC Messages 4
7.0.1 NDC API Methods Summary 8
A.1 Microphone ID Value Assignments 21

Document Revision History

Date	Ver.	Author	Description
2019-04-03	1.0.0	Alex Hogen	Final draft of API v1.0.1
2019-04-12	1.1.0	Alex Hogen	Specify SemVer for proto and FW numbers
2019-07-01	1.2.0	Alex Hogen	Update Mic ID values
2020-01-20	1.2.2	Alex Hogen	Update Mic ID values

Notice

THE INFORMATION CONTAINED IN THIS DOCUMENT IS THE SOLE PROPERTY OF AUDIX CORPORATION INCORPORATED. ANY REPRODUCTION OF THIS DOCUMENT OR THE PRODUCT(S) AND METHOD(S) DESCRIBED IN THIS DOCUMENT, IN PART OR AS A WHOLE, WITHOUT THE WRITTEN PERMISSION OF AUDIX CORPORATION, INC. IS STRICTLY PROHIBITED.

No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying and recording, for any purpose, without the express permission of Audix Corporation. Audix Corporation retains the right to make changes to this document at any time, without notice. Audix Corporation makes no warranty of any kind, expressed or implied, with regard to any information contained in this document, including, but not limited to, the implied warranties or merchantability or fitness for any particular purpose.

Further, Audix Corporation does not warrant the accuracy or completeness of the information, text, graphics, or other items contained within this document. Audix Corporation products are not designed for use in life-support equipment or applications that would cause a life-threatening situation if any such products failed. Do not use Audix Corporation products in these types of applications.

Patent(s) Pending – Products identified in this document may be covered by one or more Audix Corporation patents and/or patent applications.

Copyright © 2018-2020. Audix Corporation, Inc. All rights reserved.

1 Overview

Networked Device Control (NDC) API is designed to expose controls and information from a networked audio device, such as an Ethernet enabled microphone, to other devices in a LAN. The API messages are based on the JSON-RPC specification [1]. The messages are in ASCII text, so development is easy. Messages are kept to a minimum length so even a small embedded device can use the API.

Currently, most devices use UDP packets as the transport method, but the NDC API is transport agnostic.

Features:

- Designed to be used in simple, small UDP datagrams
- Syntax compliant with widely used JSON format
- An application-specific implementation of JSON-RPC 2.0 protocol
- Simple ASCII messages means ease of design and debugging

Limitations:

- No security
- No encryption
- No device discovery (relies on mDNS or LLMNR)
- Limited data types

For a client to begin communicating with an NDC server, it should perform the following:

1. Discover NDC server devices (Section 4)
2. Acquire control of the device (Section 6.1)
3. Enumerate device capabilities (Section 6.2)

After these steps, an NDC client can make any desired API call (Section 7). When finished talking with the server, the client should release control of the device so that other clients may connect.

2 Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC2119 when, and only when, they appear in all capitals, as shown here [2].

The NDC API uses JSON-RPC 2.0 remote procedure call (RPC) protocol for all messages (see [1]). Additional rules appended to the JSON-RPC protocol are listed in Section 5.

3 Compatibility

This is the first release of the NDC API. There are no compatibility issues between versions.

4 Server Discovery

Device discovery can be implemented in a number of ways. Currently, Audix Dante-based devices support multicast DNS (mDNS) but other mechanisms for device discovery may be added in the future. Refer to the specific Audix device's documentation to determine its method of discovery.

The default transport method and socket of a NDC server is shown in Table 4.0.1. The discovery protocol MAY override this configuration. The default is simply provided here as a reference.

Table 4.0.1: NDC Server Default Socket Configuration

Transport Method:	UDP
Port Number:	8069
Addressing Method:	Unicast

4.1 Multicast DNS (mDNS)

An NDC server MAY be discoverable with multicast DNS (mDNS). mDNS is a zero-configuration (zeroconf) service and has been implemented in software such as Apple's Bonjour [3–5].

The service name to be used in an mDNS query of an NDC server MUST be: **`_audix-ndcp._udp.local`**.

The device's reply to an mDNS service query contains at least the following information in the mDNS responses and text records:

- **"Name"**: Device name
- **"Address"**: IP(v4) address of the NDC server
- **"Port"**: UDP Port number of the NDC server
- **"Properties"**:
  - **"name"**: Friendly device name
  - **"model"**: Device model name and/or number

With this information, a client may now send a JSON-RPC message (such as `acquire_control`) addressed to the NDC server's IP address and UDP port.

5 JSON-RPC Message Rules

In addition to the JSON-RPC 2.0 specification, additional usage and formatting rules apply [1]. These restrictions provide benefits for embedded systems with limited memory. They also simplify the kinds of expected data types, allowing for optimizations in JSON parsing implementations.

```
--> {"jsonrpc":"2.0", "method":"method_name", "params":[0, 1], "id":3735928559}
```

Listing 5.0.1: JSON-RPC Request Object Example

```
<-- {"jsonrpc":"2.0", "result":"result of operation", "id":3735928559}

    // ... OR ...

<-- {"jsonrpc":"2.0", "error":{"code": -32600, "message": "Invalid Request"},
    "id":3735928559}
```

Listing 5.0.2: JSON-RPC Response Object Examples

5.1 JSON-RPC Object Formatting

JSON-RPC message formatting rules and restrictions for the NDC API, in addition to those defined in JSON-RPC 2.0, are as follows:

5.1.1. There MUST be no whitespace befor a closing or after an opening curly brace or square bracket.

Example: `{"foo"...}`

Example: `[... "bar"]`

5.1.2. There MUST be no whitespace around colons (name/value separator).

Example: `"name":"value"`

5.1.3. There MUST be no whitespace before a comma and one single space after a comma.

Example: `"vals":[1, 2, 3]`

5.2 JSON-RPC Object Usage

JSON-RPC message usage rules, in addition to those defined in JSON-RPC 2.0, are as follows:

5.2.1. The JSON-RPC message format SHALL follow the JSON-RPC 2.0 Specification [1]. Examples of NDC-formatted JSON-RPC messages are shown in Listings 5.0.1 and 5.0.2.

5.2.2. A JSON-RPC message MUST be a single JSON Object.

5.2.3. JSON-RPC Batch calls SHALL NOT be supported.

5.2.4. The "id" member of the JSON-RPC message MUST be a 32-bit unsigned decimal integer.

This field is used by the NDC API's acquisition lock. Refer to Section 6.1.

(a) The most significant 16-bits SHALL be known as "Client ID".

i. "Client ID" MUST NOT be zero (0) or 65535.

ii. "Client ID" MAY be the last two octets of the client's IPv4 address. This is not required, but can be useful when debugging.

iii. A client's "Client ID" MUST not change during the session.

(b) The least significant 16-bits SHALL be known as "Message ID".

i. "Message ID" MUST be zero (0) when a client first acquires control of a server.

ii. "Message ID" MUST NOT be the same as the "Message ID" in the last message.

iii. "Message ID" MAY be incremented by the client, serving as a message counter.

5.2.5. The following JSON data types MUST NOT be used and are unsupported.

(a) Floating-point number

(b) Hexadecimal, or any other number base besides decimal (base-10)

(c) Boolean

(d) Null

5.3 Error Responses

Section 5.1 of the JSON-RPC Specification defines a number of error codes and their meanings. It then states that the remaining error codes are "available for application defined errors" [1]. The JSON-RPC error codes and the NDC API error codes are both listed in Table 5.3.1.

Table 5.3.1: Error Codes in JSON-RPC Messages

JSON-RPC Error Codes		
Error Code	**Message**	**Meaning**
-32700	Parse error	Invalid JSON was receive by the server. An error occurred on the server while parsing the JSON text.
-32600	Invalid request	The JSON sent is not a valid Request object.
-32601	Method not found	The method does not exist / is not available.
-32602	Invalid params	Invalid method parameter(s).
-32603	Internal error	Internal JSON-RPC error.
-32000 to -32099	Server error	Reserved for implementation-defined server errors.
NDC API Error Codes		
Error Code	**Message**	**Meaning**
-32400	Invalid client/controller	The "Client ID" field of this message does not match the "Client ID" which acquired this server, OR, this server has not yet been acquired. This JSON-RPC message will be ignored.
-32450	I/O error	The server encountered a problem when interacting with an input/output device.

Note that the JSON-RPC 2.0 Specification defines "message" and "data" members of an error object. An NDC server currently has no use for the data member, so it will be omitted from error messages.

An NDC server will often include a message string in an error reply object, but it is not required. The "message" member may be omitted in memory-conservative server implementations.

6 NDC Description

The following describes the usage and operation of NDC clients and servers and how they interact with each other.

After a client discovers an NDC server (see Section 4) it will then need to acquire control of the server and enumerate the server capabilities. Then a client has all the information it needs to make NDC API calls to configure the device.

These two operations are described in Sections 6.1 and 6.2.

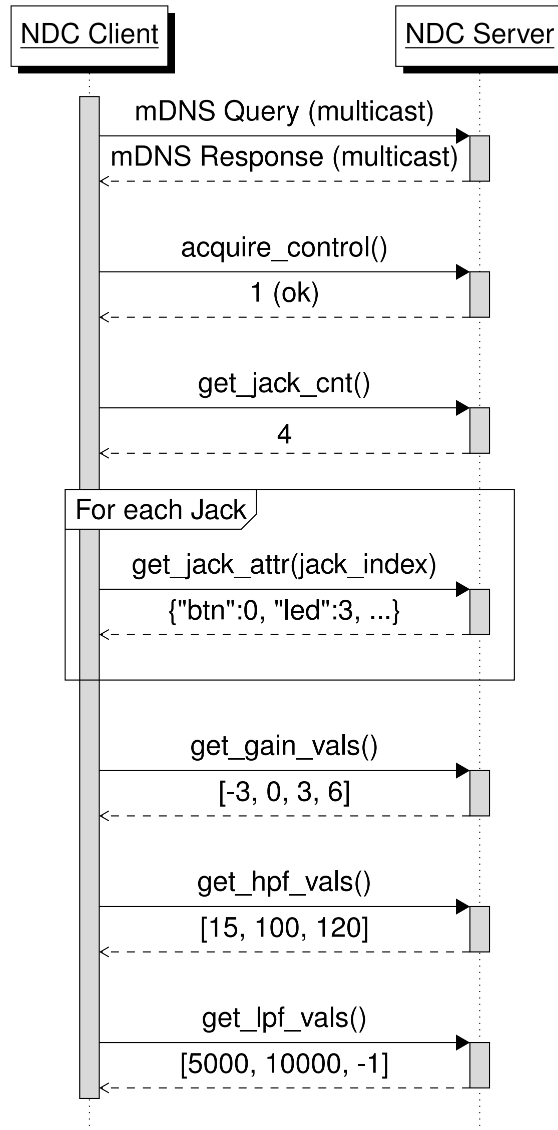

Figure 6.0.1: Common NDC client startup example

6.1 Acquire Control of NDC Server

A client must first acquire control of a server before a server will respond to other NDC API calls. As mentioned at the beginning of this document, there is no security offered by the NDC API. The NDC acquire lock is intended to prevent a client from accidentally modifying a device's settings when another client is already connected.

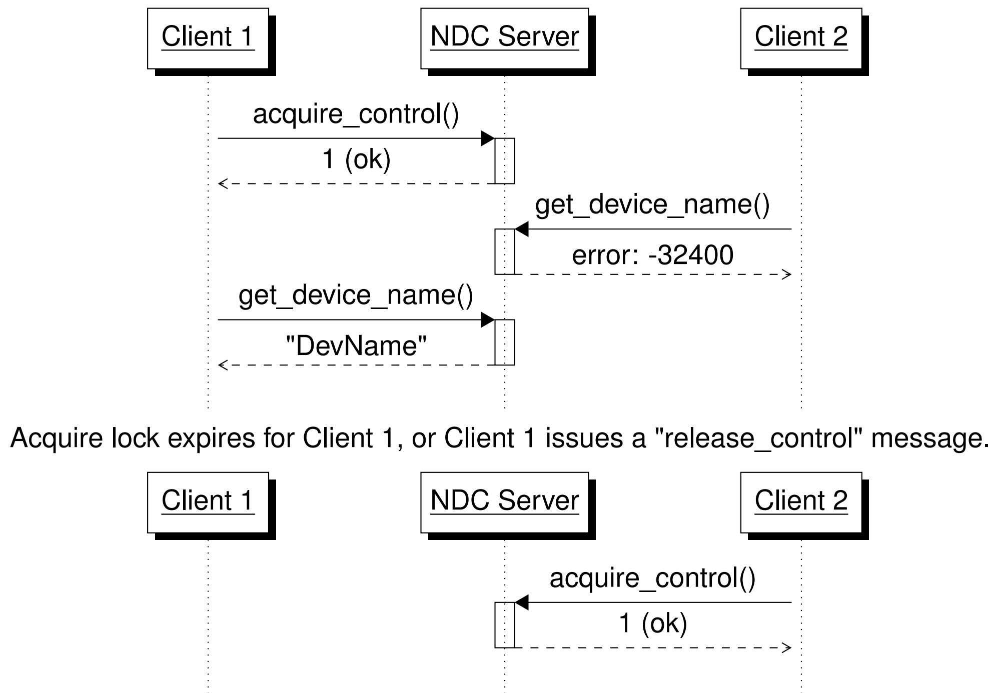

Figure 6.1.1: Acquire lock prevents secondary client communication with an acquired server

When an acquire control operation is successful, the NDC server saves the "Client ID" of that message. If any messages are received with a "Client ID" that does not match the saved value, those messages are invalid for the duration of the current session.

The acquire lock will be automatically released by the NDC server after **10 seconds** if no valid NDC API calls are made. The "Client ID" value previously saved by the server will be discarded and a new client may acquire control.

6.2 Enumeration

After discovering (Section 4) and acquiring (Section 6.1) control of a server, the client should enumerate the capabilities of the server.

The procedure for enumerating a NDC server device is outlined below and is illustrated in Figure 6.0.1.

1. Call `get_jack_cnt()` to get `JACK_CNT`
2. If `JACK_CNT` > 0, continue. Else, exit.
3. Call `get_jack_attr(jack_index)` for `JACK_CNT` iterations and record the responses.

4. If "gain" present in any Jack attributes, call `get_gain_vals()`.
5. If "hpf" present in any Jack attributes, call `get_hpf_vals()`.
6. If "lpf" present in any Jack attributes, call `get_lpf_vals()`.

The possible attributes returned by `get_jack_attr()` are documented in Section 7.7.

7 NDC Methods

This section describes each JSON-RPC method in the NDC API. The method's parameters and return types are described and example JSON-RPC request/reply messages are given.

Requests from client to server and replies from server to client are documented as shown in Listing 7.0.1.

```
--> JSON-RPC message sent from NDC Client to Server
<-- JSON-RPC message sent from NDC Server to Client
```

Listing 7.0.1: JSON-RPC example documentation syntax

Table 7.0.1 summarizes the available methods and their parameters.

Table 7.0.1: NDC API Methods Summary

Method Name	Param 1	Param 2	Param 3	Doc Section
acquire_control	-	-	-	7.1
release_control	-	-	-	7.2
get_device_name	-	-	-	7.3
get_device_firmware_ver	-	-	-	7.4
get_device_protocol_ver	-	-	-	7.5
get_jack_cnt	-	-	-	7.6
get_jack_attr	jack_index	-	-	7.7
get_mic_id	jack_index	-	-	7.8
get_btn	jack_index	btn_index	-	7.9
get_led	jack_index	led_index	-	7.10
set_led	jack_index	led_index	-	7.11
get_gain_vals	jack_index	-	-	7.12
get_gain	jack_index	gain_index	-	7.13
set_gain	jack_index	gain_index	gain_val_db	7.14
get_hpf_vals	jack_index	-	-	7.15
get_hpf	jack_index	hpf_index	-	7.16
set_hpf	jack_index	hpf_index	hpf_val_hz	7.17
get_lpf_vals	jack_index	-	-	7.18
get_lpf	jack_index	lpf_index	-	7.19
set_lpf	jack_index	lpf_index	lpf_val_hz	7.20

7.1 acquire_control

```
--> {"jsonrpc":"2.0", "method":"acquire_control", "id":<MSG_ID>}

<-- {"jsonrpc":"2.0", "result":1, "id":<MSG_ID>}
```

Listing 7.1.1: Definition of "acquire_control" request and response

Request control of the NDC server. If acquisition is successful, client will then be allowed to call other methods.

- **Parameters:**
  - None
- **Returns:**
  - `1` if acquire control successful

The `acquire_control` message must have the "Message ID" number (lower 16 bits of `id` field) set to zero (0). The upper 16 bits should be set to some unique value to identify the client. It is suggested to use the lower 16 bits of the client's IPv4 address, but any unique number is permitted.

See Section 5.2 for the definition of the `id` field. The acquire timeout time is defined in Section 6.1.

Also note that it is permissible to call `acquire_control` from a client which has already acquired the server. This could be useful to periodically reset the acquire timeout and maintain the acquire lock, if buttons are not being polled.

7.2 release_control

```
--> {"jsonrpc":"2.0", "method":"release_control", "id":<MSG_ID>}

<-- {"jsonrpc":"2.0", "result":1, "id":<MSG_ID>}
```

Listing 7.2.1: Definition of "acquire_control" request and response

Relinquish control of the NDC server.

- **Parameters:**
  - None
- **Returns:**
  - `1` if release control successful

After this method is executed, any client is able to acquire control of this server again. See `acquire_control` method in Section 7.1.

7.3 get_device_name

```
--> {"jsonrpc":"2.0", "method":"get_device_name", "id":<MSG_ID>}

<-- {"jsonrpc":"2.0", "result":"<DEVICE_NAME>", "id":<MSG_ID>}
```

Listing 7.3.1: Definition of "get_device_name" request and response

Read the name of the NDC server.

- **Parameters:**
  - None
- **Returns:**
  - Server name, as a string

A normal response from server to client returns the value as a string.

7.4 get_device_firmware_ver

```
--> {"jsonrpc":"2.0", "method":"get_device_firmware_ver", "id":<MSG_ID>}

<-- {"jsonrpc":"2.0", "result":"<DEVICE_FW_VER>", "id":<MSG_ID>}
```

Listing 7.4.1: Definition of "get_device_firmware_ver" request and response

Read the firmware version of the NDC server.

- **Parameters:**
  - None
- **Returns:**
  - Server's software/firmware version, as a string

A normal response from server to client returns the server's firmware version as a string.

The version number SHALL follow the Semantic Versioning 2.0.0 Specification [6].

7.5 get_device_protocol_ver

```
--> {"jsonrpc":"2.0", "method":"get_device_protocol_ver", "id":<MSG_ID>}

<-- {"jsonrpc":"2.0", "result":"<NDC_PROTO_VER>", "id":<MSG_ID>}
```

Listing 7.5.1: Definition of "get_device_protocol_ver" request and response

Read the NDC protocol version from the NDC server.

- **Parameters:**
  - None
- **Returns:**
  - The NDC protocol version of the Server, as a string

A normal response from server to client returns the NDC protocol version.

The version number SHALL follow the Semantic Versioning 2.0.0 Specification [6].

7.6 get_jack_cnt

```
--> {"jsonrpc":"2.0", "method":"get_jack_cnt", "id":<MSG_ID>}

<-- {"jsonrpc":"2.0", "result":<JACK_CNT>, "id":<MSG_ID>}
```

Listing 7.6.1: Definition of "get_jack_cnt" request and response

Get the number of jacks on the NDC server.

- **Parameters:**
  - None
- **Returns:**
  - The number of jacks at the Server

Many methods require a jack index parameter. This method returns the total number of jacks available. Jack indexes start at zero, so if this method returned 4, then valid jack indexes would be 0, 1, 2, and 3.

7.7 get_jack_attr

```
--> {"jsonrpc":"2.0", "method":"get_jack_attr", "params":[<JACK_INDEX>],
    "id":<MSG_ID>}

<-- {"jsonrpc":"2.0", "result":{<JACK_ATTRIBUTES>}, "id":<MSG_ID>}
```

Listing 7.7.1: Definition of "get_jack_attr" request and response

Get the attributes of a given jack.

- **Parameters:**
  1. `<JACK_INDEX>`
     Unsigned decimal, `0 >= JACK_INDEX < JACK_CNT`. See Section 7.6.
- **Returns:**
  - A JSON object, which is name/value pairs enclosed by braces. Possible attributes are:
    * **"led"** – Number of LEDs (logic outputs)
    * **"btn"** – Number of buttons/switches (logic inputs)
    * **"gain"** – Number of audio gain controls
    * **"hpf"** – Number of highpass audio filters
    * **"lpf"** – Number of lowpass audio filters

A normal response from server to client returns a JSON object containing an NDC attribute and the associated count value, as shown in Listing 7.7.1, indicating the kind and number of attributes available at the specified jack.

A attribute which is not present on a given jack may be indicated in the response message either:

- by the value in the JSON name/value pair of the attribute set to zero (0),
- OR by omitting the attribute (the JSON name/value pair) from the JSON response.

```
<-- {"jsonrpc":"2.0", "result":{"btn":4, "led":8, "gain":2, "hpf":1,
    "lpf":1}, "id":<MSG_ID>}
```

Listing 7.7.2: Example "get_jack_attr" response

7.8 get_mic_id

```
--> {"jsonrpc":"2.0", "method":"get_mic_id", "params":[<JACK_INDEX>],
    "id":<MSG_ID>}

<-- {"jsonrpc":"2.0", "result":<MIC_ID_VALUE>, "id":<MSG_ID>}
```

Listing 7.8.1: Definition of "get_mic_id" request and response

Read the microphone identification number from a given jack.

- **Parameters:**
  1. `<JACK_INDEX>`
     Unsigned decimal, `0 >= JACK_INDEX < JACK_CNT`. See Section 7.6.
- **Returns:**
  – Mic ID value, as an unsigned 8-bit type

A normal response from server to client returns an 8-bit unsigned mic ID value. This value could be converted text strings by the client using the date in Table A.1.

7.9 get_btn

```
--> {"jsonrpc":"2.0", "method":"get_btn", "params":[<JACK_INDEX>, <BTN_INDEX>],
    "id":<MSG_ID>}

<-- {"jsonrpc":"2.0", "result":<BTN_VAL>, "id":<MSG_ID>}
```

Listing 7.9.1: Definition of "get_btn" request and response

Read the status of a button logic input.

- **Parameters:**
  1. `<JACK_INDEX>`
     Unsigned decimal, `0 >= JACK_INDEX < JACK_CNT`. See Section 7.6.
  2. `<BTN_INDEX>`
     Unsigned decimal, `0 >= BTN_INDEX < BTN_CNT`. See Section 7.7.
- **Return Value:**
  – `0` means switch/button was released or OPEN
  – `1` means switch/button was pressed or CLOSED

The server will latch any pressed (CLOSED) button states until the client makes a `get_btn` request and reads that button's state. In other words, when a button is closed, the server will hold that pressed state until a client reads that specific button. The button state may be polled by the client at short, regular intervals to accurately represent the current state of the button to an end user.

7.10 get_led

```
--> {"jsonrpc":"2.0", "method":"get_led", "params":[<JACK_INDEX>, <LED_INDEX>],
    "id":<MSG_ID>}

<-- {"jsonrpc":"2.0", "result":<LED_STATE>, "id":<MSG_ID>}
```

Listing 7.10.1: Definition of "get_led" request and response

Read the current state of a single-color LED.

- **Parameters:**
  1. <JACK_INDEX>
     Unsigned decimal, 0 >= JACK_INDEX < JACK_CNT. See Section 7.6.
  2. <LED_INDEX>
     Unsigned decimal, 0 >= LED_INDEX < LED_CNT. See Section 7.7.
- **Return Value:**
  - 0 means LED is OFF
  - 1 means LED is ON

7.11 set_led

```
--> {"jsonrpc":"2.0", "method":"set_led", "params":[<JACK_INDEX>, <LED_INDEX>,
    <NEW_STATE>], "id":<MSG_ID>}

<-- {"jsonrpc":"2.0", "result":<NEW_STATE>, "id":<MSG_ID>}
```

Listing 7.11.1: Definition of "set_led" request and response

Set the state of a single-color LED.

- **Parameters:**
  1. <JACK_INDEX>
     Unsigned decimal value. 0 >= JACK_INDEX < JACK_CNT. See Section 7.6.
  2. <LED_INDEX>
     Unsigned decimal value. 0 >= LED_INDEX < LED_CNT. See Section 7.7.
  3. <NEW_STATE>
     The new LED state as an unsigned decimal value: 0 or 1.
- **Return Values:**
  - 0 means LED is not being driven
  - 1 means LED is being driven

The return value should always match <NEW_STATE> sent by the client.

7.12 get_gain_vals

```
--> {"jsonrpc":"2.0", "method":"get_gain_vals", "id":<MSG_ID>}

<-- {"jsonrpc":"2.0", "result":[<DB_VAL_0>, <DB_VAL_1>, ..., <DB_VAL_N>],
    "id":<MSG_ID>}
```

Listing 7.12.1: Definition of "get_gain_vals" request and response

Get the gain value options for all gain controls.

- **Parameters:**
  - None
- **Returns:**
  - The available gain settings as a JSON Array of signed 16-bit decimal integers.

Refer to `get_gain` and `set_gain` methods, documented in Sections 7.13 and 7.14 respectively.

Gain values are in decibels (dB). Any fractional gain value SHALL be rounded up to the nearest integer.

7.13 get_gain

```
--> {"jsonrpc":"2.0", "method":"get_gain", "params":[<JACK_INDEX>,
    <GAIN_INDEX>], "id":<MSG_ID>}

<-- {"jsonrpc":"2.0", "result":<GAIN_DB_VALUE>, "id":<MSG_ID>}
```

Listing 7.13.1: Definition of "get_gain" request and response

Read the current setting of a gain control.

- **Parameters:**
  1. `<JACK_INDEX>`
     Unsigned decimal value. 0 $>=$ `JACK_INDEX` $<$ `JACK_CNT`. See Section 7.6.
  2. `<GAIN_INDEX>`
     Unsigned decimal value. 0 $>=$ `GAIN_INDEX` $<$ `GAIN_CNT`. See Section 7.7.
- **Returns:**
  - Current gain setting (dB), as a signed 16-bit decimal integer.

The gain value returned MUST be one of the values in the array returned by `get_gain_vals` (Section 7.12).

7.14 set_gain

```
--> {"jsonrpc":"2.0", "method":"set_gain", "params":[<JACK_INDEX>,
    <GAIN_INDEX>, <GAIN_DB_VALUE>], "id":<MSG_ID>}

<-- {"jsonrpc":"2.0", "result":<GAIN_DB_VALUE>, "id":<MSG_ID>}
```

Listing 7.14.1: Definition of "set_gain" request and response

Write a new setting to a gain control.

- **Parameters:**
  1. `<JACK_INDEX>`
     Unsigned decimal value. 0 >= `JACK_INDEX` < `JACK_CNT`. See Section 7.6.
  2. `<GAIN_INDEX>`
     Unsigned decimal value. 0 >= `GAIN_INDEX` < `GAIN_CNT`. See Section 7.7.
  3. `<GAIN_DB_VALUE>`
     Signed decimal value. MUST be one of the values in the array returned by `get_gain_vals` (Section 7.12).
- **Returns:**
  - New gain setting (dB), as a signed 16-bit decimal integer.

The new gain value sent to the server is echoed back to the client in the response message, as an acknowledgment.

7.15 get_hpf_vals

```
--> {"jsonrpc":"2.0", "method":"get_hpf_vals", "id":<MSG_ID>}

<-- {"jsonrpc":"2.0", "result":[<HPF_VAL_0>, <HPF_VAL_1>, ..., <HPF_VAL_N>],
    "id":<MSG_ID>}
```

Listing 7.15.1: Definition of "get_hpf_vals" request and response

Get the highpass filter cutoff frequency options available for all highpass filter controls.

- **Parameters:**
  - None
- **Returns:**
  - The available HPF settings as a JSON Array of signed 16-bit decimal integers.

Refer to `get_hpf` and `set_hpf` methods, documented in Sections 7.16 and 7.17 respectively.

Highpass filter frequency values are in units of Hertz (Hz). Any fractional value SHALL be rounded up to the nearest integer.

NOTE: A value of negative one (-1) is the DISABLED setting. If this value is written to a given HPF, the filter will be disabled, and audio will bypass the filter, unaltered.

7.16 get_hpf

```
--> {"jsonrpc":"2.0", "method":"get_hpf", "params":[<JACK_INDEX>, <HPF_INDEX>],
    "id":<MSG_ID>}

<-- {"jsonrpc":"2.0", "result":<HPF_HZ_VALUE>, "id":<MSG_ID>}
```

Listing 7.16.1: Definition of "get_hpf" request and response

Read the current setting of a highpass filter control.

- **Parameters:**
  1. `<JACK_INDEX>`
     Unsigned decimal value. 0 $>=$ `JACK_INDEX` $<$ `JACK_CNT`. See Section 7.6.
  2. `<HPF_INDEX>`
     Unsigned decimal value. 0 $>=$ `HPF_INDEX` $<$ `HPF_CNT`. See Section 7.7.
- **Returns:**
  - Current highpass filter setting (Hz), as a signed 16-bit decimal integer.

7.17 set_hpf

```
--> {"jsonrpc":"2.0", "method":"set_hpf", "params":[<JACK_INDEX>, <HPF_INDEX>,
    <HPF_HZ_VALUE>], "id":<MSG_ID>}

<-- {"jsonrpc":"2.0", "result":<HPF_HZ_VALUE>, "id":<MSG_ID>}
```

Listing 7.17.1: Definition of "set_hpf" request and response

Write a new setting to a highpass filter control.

- **Parameters:**
  1. `<JACK_INDEX>`
     Unsigned decimal value. 0 $>=$ `JACK_INDEX` $<$ `JACK_CNT`. See Section 7.6.
  2. `<HPF_INDEX>`
     Unsigned decimal value. 0 $>=$ `HPF_INDEX` $<$ `HPF_CNT`. See Section 7.7.
  3. `<HPF_HZ_VALUE>`
     Signed decimal value. MUST be one of the values in the array returned by `get_hpf_vals` (Section 7.15).
- **Returns:**
  - New highpass filter setting (Hz), as a signed 16-bit decimal integer.

The new highpass filter value sent to the server is echoed back to the client in the response message, as an acknowledgment.

7.18 get_lpf_vals

```
--> {"jsonrpc":"2.0", "method":"get_lpf_vals", "id":<MSG_ID>}

<-- {"jsonrpc":"2.0", "result":[<LPF_VAL_0>, <LPF_VAL_1>, ..., <LPF_VAL_N>],
    "id":<MSG_ID>}
```

Listing 7.18.1: Definition of "get_lpf_vals" request and response

Get the lowpass filter cutoff frequency options available for all lowpasss filter controls.

- **Parameters:**
  - None
- **Returns:**
  - The available LPF settings as a JSON Array of signed 16-bit decimal integers.

Refer to `get_lpf` and `set_lpf` methods, documented in Sections 7.19 and 7.20 respectively.

Lowpass filter frequency values are in units of Hertz (Hz). Any fractional value SHALL be rounded up to the nearest integer.

NOTE: A value of negative one (-1) is the DISABLED setting. If this value is written to a given LPF, the filter will be disabled, and audio will bypass the filter, unaltered.

7.19 get_lpf

```
--> {"jsonrpc":"2.0", "method":"get_lpf", "params":[<JACK_INDEX>, <LPF_INDEX>],
    "id":<MSG_ID>}

<-- {"jsonrpc":"2.0", "result":<LPF_HZ_VALUE>, "id":<MSG_ID>}
```

Listing 7.19.1: Definition of "get_lpf" request and response

Read the current setting of a lowpass filter control.

- **Parameters:**
  1. `<JACK_INDEX>`
     Unsigned decimal value. 0 $>=$ `JACK_INDEX` $<$ `JACK_CNT`. See Section 7.6.
  2. `<LPF_INDEX>`
     Unsigned decimal value. 0 $>=$ `LPF_INDEX` $<$ `LPF_CNT`. See Section 7.7.
- **Returns:**
  - Current lowpass filter setting (Hz), as a signed 16-bit decimal integer.

7.20 set_lpf

```
--> {"jsonrpc":"2.0", "method":"set_lpf", "params":[<JACK_INDEX>, <LPF_INDEX>,
    <LPF_HZ_VALUE>], "id":<MSG_ID>}

<-- {"jsonrpc":"2.0", "result":<LPF_HZ_VALUE>, "id":<MSG_ID>}
```

Listing 7.20.1: Definition of "set_lpf" request and response

Write a new setting to a lowpass filter control.

- **Parameters:**
  1. `<JACK_INDEX>`
     Unsigned decimal value. 0 $>=$ `JACK_INDEX` $<$ `JACK_CNT`. See Section 7.6.
  2. `<LPF_INDEX>`
     Unsigned decimal value. 0 $>=$ `LPF_INDEX` $<$ `LPF_CNT`. See Section 7.7.
  3. `<LPF_HZ_VALUE>`
     Signed decimal value. MUST be one of the values in the array returned by `get_lpf_vals` (Section 7.18).
- **Returns:**
  - New lowpass filter setting (Hz), as a signed 16-bit decimal integer.

The new lowpass filter value sent to the server is echoed back to the client in the response message, as an acknowledgment.

Appendix

A Mic ID Values

Microphone ID values returned by the `get_mic_id` method (Section 7.8) and their meanings are shown in Table A.1. Some ID values are not yet assigned to a microphone model.

Table A.1: Microphone ID Value Assignments

ID	Microphone Type	ID	Microphone Type
1	M70	2	M55
3	-	4	M3
5	M63	6	-
7	-	8	-
9	-	10	-
11	-	12	-
13	-	14	-
15	-	16	-
17	-	18	-
19	-	20	-
21	-	22	-
23	-	24	-
25	-	26	-
27	-	28	-
29	-	30	-
31	-	32	-
33	-	34	-
35	-	36	-
37	-	38	-
39	-	40	-
255	UNKNOWN or INVALID mic		

References

[1] *JSON-RPC 2.0 Specification*, JSON-RPC Working Group, Jan. 2013. [Online]. Available: https://www.jsonrpc.org/specification (visited on 04/02/2019).

[2] S. Bradner, “Key words for use in RFCs to Indicate Requirement Levels”, RFC Editor, RFC 2119, Mar. 1997, pp. 1–2. DOI: 10.17487/RFC2119. [Online]. Available: https://tools.ietf.org/html/rfc2119.

[3] S. Cheshire and M. Krochmal, “Multicast DNS”, RFC Editor, RFC 6762, Feb. 2013, pp. 1–70. DOI: 10.17487/RFC6762. [Online]. Available: https://tools.ietf.org/html/rfc6762.

[4] S. Cheshire, Ed. (). Multicast DNS, [Online]. Available: http://www.multicastdns.org (visited on 04/03/2019).

[5] *Bonjour*, Apple Inc. [Online]. Available: https://developer.apple.com/bonjour (visited on 04/03/2019).

[6] T. Preston-Werner, *Semantic Versioning Specification (SemVer)*, version 2.0.0. [Online]. Available: https://semver.org/spec/v2.0.0.html (visited on 04/12/2019).