How to use public API interface in Genelec Smart IP devices

GENELEC®

Smart IP API Documentation – v 1, rev 0.8.4

May 2024

Table of Contents

1 Introduction 4
1.1 Using Smart IP Configurator 4
1.2 Definitions, acronyms and abbreviations 4
2 Device discovery 5
2.1 Device MAC Address 5
2.2 Discovery using mDNS query 5
2.3 Ping 5
3 Unicast API control 6
3.1 Maximum four IP connections 6
3.2 Communication format 6
3.3 Request message 6
3.3.1 Header 6
3.3.2 Body 7
3.4 Response message 7
3.4.1 Header 7
3.4.2 Body 8
3.5 Supported status codes 8
4 Smart IP device API unicast commands 9
4.1 Smart IP Manager 9
4.2 Polling frequency 9
4.3 ISS power save mode 9
4.4 Removing PoE power at IP switch 9
4.5 Device wakeup 9
4.6 API version 9
4.7 /aoip 10
4.8 /audio 11
4.9 /device 12
4.10 /events 15

4.11 /led 15
4.12 /network 16
4.13 /profile 18
5 Multicast API control 20
5.1 Multicast configuration 20
5.2 Multicast message format 20
6 Smart IP device API multicast commands 21
6.1 Volume level 21
6.2 Mute control 21
6.3 Profile selection 21
6.4 Power control 22

1 Introduction

This document describes how Genelec Smart IP devices can be controlled through the public API contained in the Smart IP devices. The API allows many aspects of a Smart IP device to be directly controlled by house automation systems and other control and management systems.

The API communication mainly uses unicast communication, while certain commands are available as multicast commands.

This API definition applies to firmware version 1.3.25 and later firmware versions.

1.1 Using Smart IP Configurator

Genelec Smart IP Configurator is a system configuration software for discovering and setting up multi-zone installed audio systems. This software can be downloaded in the MyGenelec area of Genelec web pages. For more information, look at https://www.genelec.com/smart-ip-manager.

Notice! Direct API calls to devices cannot be issued at same time when Smart IP Configurator software is in use.

1.2 Definitions, acronyms and abbreviations

These abbreviations are used in the document.

Acronym	Definition
JSON	JavaScript Object Notation
DNS-SD	DNS Service Discovery
HTTP	Hypertext Transfer Protocol
MAC address	media access control address
mDNS	multicast DNS
OUI	Organizationally unique identifier
REST	Representational state transfer
URI	Uniform Resource Identifier

2 Device discovery

2.1 Device MAC Address

Generally, manufacturer of a device can be identified by checking the first three bytes of the MAC address. These bytes form the 24-bit OUI number that identifies the vendor, manufacturer or organization manufacturing the device.

The OUI number AC-47-23 is registered to Genelec Oy.

2.2 Discovery using mDNS query

Devices can be discovered on the network using the mDNS protocol. This protocol resolves the hostnames to IP addresses within small networks that do not include a local name server.

mDNS client sends an IP protocol multicast query message that requests the hosts having a hostname to identify themselves. These devices then send a multicast message including their IP address and some other information.

Genelec devices automatically respond to the mDNS standard query after booting. mDNS response can also be requested at any time by sending the mDNS standard query '_smart_ip._tcp'.

Table 1. A sample of information in the mDNS standard query response.

Data	Content
Service	_smart_ip._tcp
Service-Instance	Genelec-00-01-22
Instance-Info	Host = 4430-000122.local Address = 192.168.0.79 Port = 9000 mac = AC:47:23:00:01:22 zoneid = 49153 zonename = living room mcastport = 49153

2.3 Ping

The Smart IP devices respond to the standard 'ping' message.

The device can be pinged:

```
ping 192.168.0.79
```

3 Unicast API control

Unicast is a method of communication where the IP network device communicates with one other IP network device.

If the same setting is needed in several IP devices, using the unicast method, the IP device must communicate to every single IP device one at a time. This leads into a sequential communication scheme. However, unicast is a very robust communication method.

3.1 Maximum four IP connections

The device has up to four simultaneous IP communication connections.

If all these IP connections are in use, additional connections cannot be established. **Therefore, it is important to close a connection that is not needed**. Especially 'keepalive' type connection will reserve a connection until keepalive time-out is reached.

Depleting the connections can hinder communication to a device.

3.2 Communication format

The Smart IP devices use REST style communication with a reduced set of HTTP/1.1 protocol. The Smart IP API only implements the HTTP methods GET and PUT and the URI can't contain a query.

Communication follows the request/response paradigm.

3.3 Request message

The request message consists of a header and an optional message body.

3.3.1 Header

The header starts with the request line having the format

```
method /public/<version>/path HTTP/1.1\r\n
```

For example, we could have e.g.

```
GET /public/v1/device/pwr HTTP/1.1\r\n
```

Then the list of request header fields follows (Table 2) and finally an empty line.

The request line and other header fields must each end to <CR><LF> (carriage return character 0x0d, line feed character 0x0a).

An empty line consists of only <CR><LF>.

The GET message usually does not need a body.

Table 2. Header fields of request HTTP message

Field name	Field value	Notes
Accept	application/json	
Connection	keep-alive	Keep connection open (default)
Connection	close	Close connection
Authorization	Basic YWRtaW46YWRtaW4=	Authentication credentials (default admin:admin)
Content-Length	0	The length of the request body
Host	192.168.0.79:9000	The domain name of server

The authorization credentials are decoded with base64 (https://www.base64decode.org/).

3.3.2 Body

The PUT message always has a header and a message body.

3.4 Response message

The response message consists of a header and an optional message body.

3.4.1 Header

The response header starts with a status line containing the status code and the reason message, for example

```
HTTP/1.1 200 OK\r\n
```

This is followed by a list of request header fields (see Table 3) and an empty line.

The request line and other header fields must each end to <CR><LF>.

The empty line must consist of only <CR><LF>.

<CR> or carriage return character contains the value 0x0d, and the <LF> line feed character 0x0a.

Table 3. Header fields of response HTTP message

Field name	Field value	Notes
Content-Type	application/json	
Connection	keep-alive	Keep connection open (default)
Connection	close	Close connection
Content-Length	0	The length of the response message body.
WWW-Authenticate	Basic	Indicates the authentication scheme that should be used

3.4.2 Body

Message body is commonly in the human readable JSON format (see http://json.org/) consisting of attribute-value pairs and array data types.

The official MIME type of JSON is 'application/json'.

Table 4. Supported JSON data types.

Data type	Values
string	A sequence of zero or more characters (ASCII). Strings are delimited with double quotation.
number	A signed decimal number (32-bit integer or float).
boolean	true or false
array	An ordered list of zero or more values of any type. Square bracket notation with comma separated elements.

An example of the JSON representation:

```
{
      "ip": "192.168.0.79",
      "port": 9000,
      "versions": ["v1", "v2"]
}
```

3.5 Supported status codes

Responses from the Smart IP device contain status codes to inform the API client about the success of the operation.

Table 5. The list of supported status codes.

Code	Status	Explanation
200	OK	Successful operation.
400	Bad Request	Invalid request, syntax error, etc
401	Unauthorized	Authentication is required and has failed or has not yet been provided.
404	Not Found	The requested resource could not be found.
405	Method Not Allowed	The requested method is not supported.
500	Internal Server Error	Unexpected condition was encountered (out of memory, failed operation etc).
503	Server Unavailable	The server cannot handle the request.

4 Smart IP device API unicast commands

4.1 Smart IP Manager

Notice! Do not use Smart IP API at same time with Smart IP Manager software.

The Smart IP Manager is a configuration software for efficiently setting up a system of Smart IP devices on an IP network. However, Smart IP Manager software cannot be used simultaneously with API commands and there is no mechanism to automatically update the state known to multiple software simultaneously controlling certain Smart IP device, and the communication can fall out of synchronization. Therefore, it is advisable to use either Smart IP Manager or the API commands, but not both simultaneously.

4.2 Polling frequency

Too frequent polling of a Smart IP device can flood the IP network and overload the device IP network interface, and these can reduce the loudspeaker responsiveness on the IP network. It is advisable to keep the message frequency as low as possible and avoid continuous polling.

4.3 ISS power save mode

The device can be powered down with a command. Powering down a device can save electrical power while the IP network interface continues to run and the device is responding to network communication. See ‘ISS Sleep’ and ‘Standby’ states for more details.

4.4 Removing PoE power at IP switch

However, if a Smart IP device is not powered, it will not respond on the IP network. Smart IP devices are not powered if the power supply available via the PoE from an IP switch device connecting to the Smart IP device is removed. This is usually possible by controlling the IP switch device. When not powered, the Smart IP devices will require some time to boot before they start responding to API calls and pass audio. This time depends on the specific Smart IP device, but can be up to one minute.

4.5 Device wakeup

Device will not respond to all command during ISS_SLEEP and STANDBY state. Device can always be woken up by sending "state"="ACTIVE" with /device/pwr command.

4.6 API version

This command reports the API version in a Smart IP device.

```
{ip}:{port}/public/{versionstring}/
```

- **ip**: *required (string),* IP address (e.g. 192.168.0.79).
- **port**: *required (string).* IP port number (default 9000).

- **versionstring**: *required (string)*, API version string e.g. v1

API version can be read by sending request

```
GET {ip}:{port}/device/info.
```

4.7 /aoip

Get identification of AoIP module.

```
GET /aoip/dante/identity
```

Media type: application/json

Properties:

- **id**: *required(string)*, Identification string
- **name**: *required(string)*, Name that is used as network host name
- **fname**: *required(string)*, Friendly name
- **mac**: *required(string)*, Ethernet MAC-address
- **locked**: *(boolean)*, true - if Dante configuration is locked.

Example:

```
{
    "id": "001dc1fffe85ad99",
    "name": "Genelec-85ad99",
    "fname": "Genelec-85ad99",
    "mac": "00:1D:C1:85:AD:99"
}
```

Get network settings of AoIP module.

```
GET /aoip/ipv4
```

Media type: application/json

Properties:

- **ip**: *required(string)*, IP address
- **mask**: *required(string)*, Mask
- **gw**: *required(string)*, Gateway

Example:

```
{
    "ip": "172.16.0.64",
    "mask": "255.255.255.0",
    "gw": "172.16.0.1"
}
```

4.8 /audio

These commands control the audio input used, output level and the mute of the Smart IP device.

Get list of selected inputs

```
GET /audio/inputs
```

Select inputs.

```
PUT /audio/inputs
```

Media type: application/json

Properties:

- **input**: *required (array of)*

  A: analog input connector
  AoIP01: AoIP input channel 1
  AoIP02: AoIP input channel 2

Example:

```
{
  "input": [
    "AoIP01",
    "AoIP02"
  ]
}
```

Get loudspeaker level and mute state.

```
GET /audio/volume
```

Set loudspeaker level and mute.

```
PUT /audio/volume
```

Media type: application/json

Properties:

- **level**: *(number - minimum: -200 - maximum: 0),* Volume level in 0.1 dB resolution.
- **mute**: *(boolean),* Mute audio.

Example:

```
{
  "level": -5.2,
  "mute": false
}
```

4.9 /device

This command reports data on the device.

Returns device information stored in permanent OTP flash memory like serial number, MAC address etc.

```
GET /device/id
```

Media type: application/json

Properties:

- **barcode**: *required (string - minLength: 7 - maxLength: 20),*
  Bar code value. Defined during production.
- **mac**: *required (string - maxLength: 17),*
  MAC address. Defined during production.
- **hwId**: *required (string - maxLength: 32),*
  Hardware version number in format major.minor.rev.build. Set up during production.
- **model**: *required (string - maxLength: 32),*
  Device model name. Defined during production. This is display name that has no functions in the device side.
- **modId**: *required (string - maxLength: 32),*
  Model specific configuration. Defined during production. This parameter defines whitch configuration set device is using.

Example:

```
{
  "barcode": "4430-123456",
  "mac": "AC:47:23:01:02:03",
  "hwId": "",
  "model": "4430",
  "modId": "4430-1"
}
```

Get API version string, model name and a set of version information.

```
GET /device/info
```

Media type: application/json

Properties:

- **fwId**: *(string)*
  Firmware identification number in format model_base-major.minor.rev-build_date_and_time.
- **build**: *(string)*
  Committed GIT revision number. -modif means that uncommitted source code is used when creating firmware.
- **baseId**: *(string)*
  Platform software version number in format major.minor.rev. This represents the common source code in folder name srcc.
- **hwId**: *(string)*
  Hardware version string. Set up during production.
- **model**: *(string)*
  Device model name. Set up during production.
- **category**: *(string)*
  SAM_1W, SAM_2W, SAM_3W, MICR.
- **technology**: *(string)*
  SAM_IP
- **upgradeId**: *(integer)*
  Compability information for upgrading firmware
- **apiVer**: *(string)*
  API version
- **confirmFwUpdate**: *(boolean)*
  New firmware is running and waiting for confirmation from user. Bootloader reverts backup firmware during next reboot if confirmation is not done.

Example:


```
{
  "fwId": "44x0-1-0-5-201903121011",
  "build": "6ef154-modif",
  "baseId": "1.0.0",
  "hwId": "304-4430 revA",
  "upgradeId": 10,
  "model": "4430",
  "category": "SAM_2W",
  "technology": "SAM_IP",
  "apiVer": "v1"
}
```

Switch between sleep and active state. Device can be booted also.

```
PUT /device/pwr
```

Media type: application/json

Properties:

- **state**: *(one of STANDBY, ACTIVE, BOOT, AOIPBOOT)*

Example:

```
{
  "state": "STANDBY"
}
```

Get power state.

```
GET /device/pwr
```

Media type: application/json

Properties:

- **state**: *(one of STANDBY, ACTIVE, ISS_SLEEP, PWR_FAIL)*
- **poeAllocatedPwr**: *(number)*
- Power allocated by PoE PSE (switch) [0.1W].
- **poePd15W**: *(boolean)*
- true - if PoE PD (loudspeaker) limits current consumption to 15W, false - if full power is needed (30W).

Example:

```
{
  "state": "ACTIVE",
  "poeAllocatedPwr": 25.5,
  "poePd15W": false
}
```

4.10 /events

Read measurement data.

```
GET /events
```

Media type: application/json

Properties:

- **bsLevel**: *(number)*, Bass output level.
- **twLevel**: *(number)*, Tweeter output level.
- **inLevel**: *(number)*, input level.
- **cpuT**: *(number)*, CPU temperature.
- **nwInKbps**: *(number)*, Network traffic to host CPU [kbps].
- **cpuLoad**: *(integer)*, CPU load.
- **uptime**: *(string)*, Time from startup.

Example:

```
{
  "bsLevel": -125.9,
  "twLevel": -110.6,
  "inLevel": -116.1,
  "cpuT": 51,
  "nwInKbps": 0,
  "cpuLoad": 73,
  "uptime": "3d 10h 36m 12s"
}
```

4.11 /led

This command controls the front panel LED lights and how it is used.

Control led visibility

```
PUT /device/led
```

Get led settings

```
GET /device/led
```

Media type: application/json

Properties:

- **ledIntensity**: (number), Led brightness [0, 100]%.
- **rj45Leds**: (Boolean),
  true – RJ45 ethernet connector leds are enabled
  false - RJ45 ethernet connector leds are disabled
- **hideClip**: (Boolean)
  true – clip led is disabled
  false – clip led is enabled

Hide clip property is applicable to subwoofer only.

Example:

```
{
    "ledIntensity": 70,
    "rj45Leds": true,
    "hideClip": false
}
```

4.12 /network

Network related endpoints.

Read network configuration.

```
GET /network/ipv4
```

Write network configuration.

```
PUT /network/ipv4
```

Media type: application/json

Properties:

- **hostname**: *(string - minLength: 1 - maxLength: 63)*
  Valid characters [A-Z], [a-z], [0-9], -
- **mode**: *(one of auto, static)*
  auto - DHCP or autoip active, static - static ip set by user
- **ip**: *(string)*
  IP address. Not needed if mode is auto.
- **mask**: *(string)*
  IP address mask. Not needed if mode is auto.
- **gw**: *(string)*
  Gateway. Not needed if mode is auto.

- **volIp**: *(string)*
  Group address for multicast control. Set address 0.0.0.0 to prevent device joining to multicast group.
- **volPort**: *(integer - minimum: 1024 - maximum: 65535)*
  Port number for multicast control.
- **auth**: *(string - maxLength: 64)*
  Credentials in format user:passwd.

Example:

```
{
  "hostname": "4430-000102",
  "mode": "static",
  "ip": "192.168.0.1",
  "mask": "255.255.255.0",
  "gw": "192.168.0.100",
  "volIp": "239.0.0.2",
  "volPort": 49152,
  "auth": "admin:admin"
}
```

Get zone info.

```
GET /network/zone
```

Media type: application/json

Properties:

- **zone**: *required (integer - minimum: 0)*
  Zone identification. Device can be removed from zone by sending zone id = 0 or by sending zone id with empty name.
- **name**: *required (string - maxLength: 50)*
  Name of the zone.

Example:

```
{
  "zone": 2,
  "name": "Listening room"
}
```

4.13 /profile

Endpoints for controlling profiles.

Get list of profiles stored in device.

```
GET /profile/list
```

Media type: application/json

Properties:

- **selected**: *required (integer - minimum: 0 - maximum: 5)*
  Current profile ID selected by user. Notice! The device will use default settings (profile ID = 0) in case of selected profile ID is not stored to device.
- **startup**: *required (integer - minimum: 0 - maximum: 5)*
  Startup defines profile ID which will be selected after reboot. Notice! The device will use default settings (profile ID = 0) in case of selected profile ID is not stored to device.
- **list**: *required (array of profileitem)*
  List of stored profiles.
  - **id**: *required (integer - minimum: 0 - maximum: 5)*
    Profile ID. Number of profiles in each zone and device is limited to 5. Profile id 0 stands for default settings.
  - **name**: *required (string - maxLength: 50)*
    Name of the profile.

Example:

```
{
  "selected": 0,
  "startup": 0,
  "list": [
    {
      "id": 5,
      "name": "Profile 5"
    }
  ]
}
```

Restore profile from flash and set it as an active profile.

```
PUT /profile/restore
```

Media type: application/json

Properties:

- **id**: *required (integer - minimum: 0 - maximum: 5)*
  Profile ID to be restored. Notice! The device will use default settings (profile ID = 0) in case of selected profile ID is not stored to device.
- **startup**: *(boolean)*
  If true; profile ID =is selected as an active profile after power reset. Notice! The device will use default settings (profile ID = 0) in case of selected profile ID is not stored to device.

Example:

```
{
  "id": 1,
  "startup": false
}
```

5 Multicast API control

Multicast is a group communication method where one message is addressed to a group of devices simultaneously. This makes communication faster, improves synchronization of command actions, and reduces traffic on the IP network.

Multicast messages spread across the IP network, and this can lead to load on the network. To reduce multicast messages in the network, it is recommended to enable IGMP snooping on routers.

Multicast communication is less secure to unicast communication as the multicast recipients do not interact with the IP message sender to ensure that message is correctly received or received at all, and the sender of multicast message may remain ignorant of the success of the message.

5.1 Multicast configuration

To use multicast, a device must join a multicast group.

The device will join a multicast group automatically if the group IP address (224.0.0.0 – 239.255.255.255) and port number (49152 – 65535) is set in the device. Setting can be done with Smart IP Manager.

Multicast control can be disabled by setting multicast address to 0.0.0.0.

5.2 Multicast message format

Every multicast message starts with JSON name “mcast” which has message object. Message object contains version info and message name/value pair.

Empty message looks like this:

```
{
  "mcast": {
    "ver": 1
  }
}
```

6 Smart IP device API multicast commands

6.1 Volume level

Properties:

- **level**: *required (float),* Volume level range is [-130.0, 0.0]

Example:

```
{
  "mcast": {
    "ver": 1,
    "level": -30
  }
}
```

6.2 Mute control

Properties:

- **mute**: *required (boolean),*
  Muted device is indicated with orange front led and unmuted with green front led.

Example:

```
{
  "mcast": {
    "ver": 1,
    "mute": true
  }
}
```

6.3 Profile selection

Smart IP devices can keep five complete configurations in memory. These configurations are called profiles. The profile selection enables the profiles to be recalled for use.

Properties:

- **profile**: *required (integer)*
  The profile can be changed by sending the profile number. Number 0 is the default profile with flat response. Profiles 1-5 can be setup with Smart IP Manager.

Example:

```
{
  "mcast": {
    "ver": 1,
    "profile": 0
  }
}
```

6.4 Power control

This command can efficiently turn on and off, as well as boot the complete multicast set of Smart IP devices.

Properties:

- **state**: *required (text)*
  The device can be put to standby mode by using value “STANDBY” and wake up by booting device with value “BOOT”.

Examples:

```
{
  "mcast": {
    "ver": 1,
    "state": "STANDBY"
  }
}
```

```
{
  "mcast": {
    "ver": 1,
    "state": "BOOT"
  }
}
```

(end of document)