XSCACE

CONTROLLER

Air Mini IP Control

Every Air Mini and Air Amp answers plain HTTP on your LAN. No cloud, no bridge, no vendor SDK — a single GET request per command, which means any control system that can fire a URL can drive the speaker.

FIRMWARE EC04 / A33M | PORT 8000 | HTTP GET | REV 2026-08

CONTENTS — CLICK TO JUMP

Every command below is printed as a complete URL, ready to copy. Replace `<device-ip>` with the speaker's own address — nothing else needs changing. From a terminal, wrap it: `curl "…"`

Canadian Audio Labs Private Limited · xscace.com

01 — QUICK START

CONTENTS ↑

Set the volume to 40 on a speaker. That is the whole protocol: one URL, one response.

```
# set volume to 40
curl "http://<device-ip>:8000/?Instruct=setPlayerCmd:vol:40"
→ OK

# what is it doing right now?
curl "http://<device-ip>:8000/?Instruct=getPlayerStatus"
→ {"PlaySource":"USB Disk","PlayStatus":"play","Volume":"40","Mute":"unmute"}
```

> FINDING A SPEAKER
>
> Speakers advertise over mDNS, but the reliable method for a fixed install is to give each one a static IP (section 12) and address it directly. DHCP re-leases move speakers — we have watched one jump from `.122` to `.74` after a firmware reboot, which breaks every driver addressing it by IP.

02 — HOW REQUESTS WORK

CONTENTS ↑

One pattern covers the entire API.

```
http://<device-ip>:8000/?Instruct={command}
https://<device-ip>:8443/?Instruct={command}   ← same commands over TLS
```

RESPONSE	MEANS
`OK`	Command accepted.
`Not`	Command not implemented, or not valid for the current source. It is a bare string, not JSON — parse defensively.
`FAIL`	Command understood but rejected, usually a bad argument.
`JSON`	Query commands return an object. Values are strings, including numbers — `"Volume":"40"`, not `40`.

> DO NOT BLANKET-ENCODE
>
> Most commands are safe to URL-encode. Parametric EQ is not — its JSON body must reach the firmware raw, with real `{ } " [ ]` characters. See section 09.

03 — READING THE STATUS MARKS

CONTENTS ↑

The manufacturer's documentation lists commands the firmware never implemented. Every command here carries a mark based on what we measured on real hardware, so you do not lose an afternoon to a command that was never going to answer.

WORKS	**Confirmed on hardware** Tested on CL-BOPro_XSCACE units running 2.34.0023.33. Safe to build on.
CAVEAT	**Works, with a catch** Functional, but behaves in a way that will surprise you. Read the note before using it.
NOT IMPLEMENTED	**Documented but dead** Present in the manufacturer's manual, absent from the firmware. Returns `Not` or HTTP 400. Do not design around it.

04 — DEVICE INFO

CONTENTS ↑

Three read commands cover everything a control system needs to display.

getStatusEx

WORKS

```
http://<device-ip>:8000/?Instruct=getStatusEx
```

Full device status as JSON — UUID, name, IP, firmware version, EQ state, and the `DevFunction` array listing the inputs this unit actually has.

getPlayerStatus

WORKS

```
http://<device-ip>:8000/?Instruct=getPlayerStatus
```

Playback state — `PlaySource`, `PlayStatus`, `Volume`, `Mute`, `LoopMode`. This is your polling command.

getMetaInfo

WORKS

```
http://<device-ip>:8000/?Instruct=getMetaInfo
```

Now playing — `Title`, `Artist`, `Album`, plus `Schedule` (elapsed seconds) and `Totlen` (duration). Returns the bare string `Not` when nothing is loaded.

> **BUILDING A PROGRESS BAR**
>
> `Schedule` is the playhead. It reads `"110s"` for 110 seconds elapsed, advances one second per second while playing, and holds still while paused. With `Totlen` that is everything a progress bar needs.
>
> Poll every few seconds and tick locally in between — but drive your local clock from the arrival of a poll, not from the value changing. A stalled stream reports the same `Schedule` while still claiming to play, and a bar keyed on the value will run to the end of a track that is not moving.

> **TWO NAMES FOR ONE THING**
>
> `getStatusEx` returns `PlayState` with the value `"plays"`. `getPlayerStatus` returns `PlayStatus` with the value `"play"`. Same state, different field, different spelling. Normalise both.

setPlayerCmd:play:{url}

WORKS

```
http://<device-ip>:8000/?Instruct=setPlayerCmd:play:{url}
```

Play a stream URL. Accepts a JSON body for metadata: `{"Url":"…","Title":"…","Artist":"…"}`.

setPlayerCmd:pause

WORKS

```
http://<device-ip>:8000/?Instruct=setPlayerCmd:pause
```

Pause.

setPlayerCmd:resume

WORKS

```
http://<device-ip>:8000/?Instruct=setPlayerCmd:resume
```

Resume.

setPlayerCmd:onepause

WORKS

```
http://<device-ip>:8000/?Instruct=setPlayerCmd:onepause
```

Toggle play and pause — one button on a keypad.

setPlayerCmd:stop

WORKS

```
http://<device-ip>:8000/?Instruct=setPlayerCmd:stop
```

Stop.

setPlayerCmd:prev

WORKS

```
http://<device-ip>:8000/?Instruct=setPlayerCmd:prev
```

Previous track.

setPlayerCmd:next

WORKS

```
http://<device-ip>:8000/?Instruct=setPlayerCmd:next
```

Next track.

setPlayerCmd:setplay:{seconds}

CAVEAT

```
http://<device-ip>:8000/?Instruct=setPlayerCmd:setplay:{seconds}
```

Seek. Returns `OK` and the playhead really moves — but see the note below about seeking while paused.

setPlayerCmd:getplay:seconds

NOT IMPLEMENTED

```
http://<device-ip>:8000/?Instruct=setPlayerCmd:getplay:seconds
```

Returns `Not` / HTTP 400. Read `Schedule` from `getMetaInfo` instead — it gives you the same answer.

setPlayerCmd:loopmode:{1-5}

WORKS

```
http://<device-ip>:8000/?Instruct=setPlayerCmd:loopmode:{1-5}
```

`1` single · `2` list · `3` no loop · `4` random loop · `5` random, no loop.

> **SEEKING WHILE PAUSED IS QUEUED, NOT APPLIED**
>
> Send `setplay` to a paused speaker and it returns `OK`, but nothing moves — `Schedule` does not change. The seek is held and applied the moment playback resumes, at which point the position jumps.
>
> If your UI reads back the position to confirm a seek, it will look like the command failed. Hold your own value until playback resumes.

> **SPOTIFY CONNECT OWNS ITS OWN TRANSPORT**
>
> When the source is Spotify Connect, `pause`, `resume` and `onepause` all return `Not`. Spotify controls playback; the speaker will not override it. Volume still works.

06 — VOLUME

CONTENTS ↑

setPlayerCmd:vol:{0-100}

WORKS

```
http://<device-ip>:8000/?Instruct=setPlayerCmd:vol:{0-100}
```

Absolute volume.

setPlayerCmd:RemoteVol++

WORKS

```
http://<device-ip>:8000/?Instruct=setPlayerCmd:RemoteVol++
```

Step up — good for a hold-to-repeat keypad button.

setPlayerCmd:RemoteVol--

WORKS

```
http://<device-ip>:8000/?Instruct=setPlayerCmd:RemoteVol--
```

Step down.

setPlayerCmd:mute:1 WORKS

```
http://<device-ip>:8000/?Instruct=setPlayerCmd:mute:1
```

Mute.

setPlayerCmd:mute:0 WORKS

```
http://<device-ip>:8000/?Instruct=setPlayerCmd:mute:0
```

Unmute.

setPlayerCmd:maximumVolume:{10-100} NOT IMPLEMENTED

```
http://<device-ip>:8000/?Instruct=setPlayerCmd:maximumVolume:{10-100}
```

Returns `Not`; `DevVolumeMax` never changes. There is no handler in the firmware. Enforce a volume ceiling in your control system, not on the speaker.

07 — SOURCES

CONTENTS ↑

Switching inputs is one command — but the token you send is not the one the manual documents.

> SEND THE DISPLAY STRING, NOT THE MANUAL'S TOKEN
>
> The manual lists tokens like `USBDisk`, `LineIn` and `HDMIARC`. Real firmware rejects all of them with `FAIL`.
>
> What it accepts is the human-readable string the unit reports in the `DevFunction` array of `getStatusEx` — with the spaces: `USB Disk`, `AUX In`, `HDMI ARC`. Percent-encode the space in the URL; the firmware decodes it.
>
> **Read `DevFunction` first and switch to the exact strings it gives you.** Inputs vary between an Air Mini and an Air Amp — do not hard-code a list.

```
# right — the display string from DevFunction
…/?Instruct=setPlayerCmd:switchmode:USB%20Disk     → OK

# wrong — the documented token
…/?Instruct=setPlayerCmd:switchmode:USBDisk        → FAIL
```

setPlayerCmd:switchmode:Network WORKS

```
http://<device-ip>:8000/?Instruct=setPlayerCmd:switchmode:Network
```

Network streaming. This token is the same either way.

setPlayerCmd:switchmode:Bluetooth WORKS

```
http://<device-ip>:8000/?Instruct=setPlayerCmd:switchmode:Bluetooth
```

Bluetooth.

setPlayerCmd:switchmode:USB%20Disk WORKS

```
http://<device-ip>:8000/?Instruct=setPlayerCmd:switchmode:USB%20Disk
```

USB Disk — note the percent-encoded space. Confirm the string in `DevFunction` before relying on it.

setPlayerCmd:switchmode:AUX%20In CAVEAT

```
http://<device-ip>:8000/?Instruct=setPlayerCmd:switchmode:AUX%20In
```

AUX In, if this unit has it. Take the exact string from `DevFunction`.

setPlayerCmd:switchmode:HDMI%20ARC CAVEAT

```
http://<device-ip>:8000/?Instruct=setPlayerCmd:switchmode:HDMI%20ARC
```

HDMI ARC, if fitted. Same rule — read `DevFunction` first.

setPlayerCmd:switchmode:{DevFunction string} CAVEAT

```
http://<device-ip>:8000/?Instruct=setPlayerCmd:switchmode:{DevFunction string}
```

Every other input — `Optical In`, `RCA In`, `PHONO In`. Percent-encode the spaces.

08 — STEREO PAIRING

CONTENTS ↑

Two Air Minis become a stereo pair by assigning each one a channel. All three commands are confirmed on hardware.

setChannel:ll WORKS

```
http://<device-ip>:8000/?Instruct=setChannel:ll
```

Left channel only.

setChannel:rr WORKS

```
http://<device-ip>:8000/?Instruct=setChannel:rr
```

Right channel only.

setChannel:rl WORKS

```
http://<device-ip>:8000/?Instruct=setChannel:rl
```

Both channels — the default, and how you undo a pair.

> ORDER OF OPERATIONS
>
> Group the two speakers first (section 10), then assign channels. A stereo pair is a multiroom group where the host is `ll` and the slave is `rr`.

09 — EQ

CONTENTS ↑

Three EQ engines: bass/treble, ten-band preset, and ten-band parametric. Enable the engine first, then send the values.

Enabling an engine

EQENABLE:	ENGINE
0000	Off
0001	Bass / treble
0010	Preset
0011	Bass / treble + preset
0100	Parametric
0101	Bass / treble + parametric

0110 and 0111 are not supported — preset and parametric cannot run at the same time. Enable an engine with http://<device-ip>:8000/?Instruct=eqEnable:0100.

setEqHighAndLowFrequencies:{"Bass":"3","Treble":"-2"} WORKS

```
http://<device-ip>:8000/?Instruct=setEqHighAndLowFrequencies:{"Bass":"3","Treble":"-2"}
```

Bass and treble, each −5 … 5. Requires `eqEnable:0001` first.

eqEnable:0100 WORKS

```
http://<device-ip>:8000/?Instruct=eqEnable:0100
```

Enable the parametric engine. Wait roughly 500 ms before sending band data.

`setParameterEq:{json}` CAVEAT

```
http://<device-ip>:8000/?Instruct=setParameterEq:{json}
```

The ten bands. See the block below for the shape.

> **Do not send this through a normal HTTP client.** The JSON must reach the firmware unencoded — use raw TCP to port 8000.

`getEqInfo:1` WORKS

```
http://<device-ip>:8000/?Instruct=getEqInfo:1
```

Read bass / treble values.

`getEqInfo:3` WORKS

```
http://<device-ip>:8000/?Instruct=getEqInfo:3
```

Read the current parametric bands.

`getEqSwitch` WORKS

```
http://<device-ip>:8000/?Instruct=getEqSwitch
```

Which engine is on.

`addCustomEqInfo:{json}` WORKS

```
http://<device-ip>:8000/?Instruct=addCustomEqInfo:{json}
```

Store a named preset on the speaker. `"Type":"2"` preset, `"Type":"3"` parametric.

`delCustomEqInfo:{json}` WORKS

```
http://<device-ip>:8000/?Instruct=delCustomEqInfo:{json}
```

Delete a stored preset by name.

Parametric — the full body

```
# 1. enable the engine    2. wait 500 ms    3. send the bands
eqEnable:0100
setParameterEq:{"Custom":"1","Name":"custom","EqBand":[
    {"Index":"0","Filter":"PK","Freq":"100.0","Q":"0.707","Gain":"3.0"}, … ×10 ]}
```

FIELD	RANGE
Filter	PK · LS · HS · LP · HP · OFF
Freq	20.0 – 24000.0 Hz
Q	0.0 – 30.0
Gain	−12.0 – 12.0 dB

> THE ONE THAT COSTS PEOPLE A DAY
>
> Parametric EQ JSON **must not be URL-encoded**. Most HTTP clients percent-encode `{ } " [ ]` automatically, and the firmware rejects the result outright.
>
> Use a raw TCP socket, or an HTTP client you can stop from encoding the query string. Our own app had to drop to raw TCP for exactly this.
>
> And always `eqEnable:0100` **before** the band data, with roughly 500 ms between them. Send the bands first and they are silently discarded.

10 — MULTIROOM

CONTENTS ↑

One speaker becomes the host; others join it as slaves. Always set the host first.

multiroom:setHost

WORKS

```
http://<device-ip>:8000/?Instruct=multiroom:setHost
```

Make this speaker the host. Do this before adding any slave.

multiroom:setSlave:{host_ip}

WORKS

```
http://<device-ip>:8000/?Instruct=multiroom:setSlave:{host_ip}
```

Sent to the slave, naming the host's IP.

multiroom:disconnectSlave

WORKS

```
http://<device-ip>:8000/?Instruct=multiroom:disconnectSlave
```

Sent to the slave — removes just that one.

multiroom:breakUp

WORKS

```
http://<device-ip>:8000/?Instruct=multiroom:breakUp
```

Sent to the host — dissolves the whole group.

multiroom:getInformation WORKS

```
http://<device-ip>:8000/?Instruct=multiroom:getInformation
```

Current group state as JSON.

RESPONSE	MEANING
`HostIp = MyIp`	This speaker is the host.
`HostIp ≠ MyIp`	This speaker is a slave of `HostIp`.
`Host = "0"`	Not grouped.

11 — USB PLAYBACK

CONTENTS ↑

getUsbSongList WORKS

```
http://<device-ip>:8000/?Instruct=getUsbSongList
```

Track index as JSON: `{"1":"/path/file.flac", …}`.

selectUsbTracks:{n} WORKS

```
http://<device-ip>:8000/?Instruct=selectUsbTracks:{n}
```

Play track `n` from that list.

playFromUsbDisk WORKS

```
http://<device-ip>:8000/?Instruct=playFromUsbDisk
```

Start USB playback.

> NEVER CACHE TRACK NUMBERS
>
> The index belongs to the drive as the speaker currently sees it. Swap the stick, add a file, or re-mount, and every number shifts. A cached number plays a different song.
>
> Re-read `getUsbSongList` each time you present the list.

12 — NETWORK & STATIC IP

CONTENTS ↑

For any fixed install, pin the address. This is the single most valuable command in the document for an integrator.

setNetIPSwitchState:Enable WORKS

```
http://<device-ip>:8000/?Instruct=setNetIPSwitchState:Enable
```

Static IP on both interfaces.

setNetIPSwitchState:E_Enable WORKS

```
http://<device-ip>:8000/?Instruct=setNetIPSwitchState:E_Enable
```

Ethernet only.

setNetIPSwitchState:W_Enable WORKS

```
http://<device-ip>:8000/?Instruct=setNetIPSwitchState:W_Enable
```

Wi-Fi only.

setNetIPSwitchState:Disable WORKS

```
http://<device-ip>:8000/?Instruct=setNetIPSwitchState:Disable
```

Back to DHCP.

setStaticIP:{json} WORKS

```
http://<device-ip>:8000/?Instruct=setStaticIP:{json}
```

The address itself — `W` fields for Wi-Fi, `E` for Ethernet. Full example below.

wlanGetConnectState WORKS

```
http://<device-ip>:8000/?Instruct=wlanGetConnectState
```

`OK` · `FAIL` · `PROCESS`.

connectToAP:wifiName:{ssid}:wifiPassword:{pass} CAVEAT

```
http://<device-ip>:8000/?Instruct=connectToAP:wifiName:{ssid}:wifiPassword:{pass}
```

Join a network. Use `%23` for any `#` in the SSID or password.

Static IP — the full body

```
http://<device-ip>:8000/?Instruct=setStaticIP:{"WStaticIp":"192.168.1.52",
"WStaticNetmask":"255.255.255.0","WStaticGateway":"192.168.1.1","WStaticDns":"192.168.1.1"}
```

SILENCE HERE MEANS SUCCESS

Applying a static IP changes the address you are talking to, so the reply often never arrives. Treat a timeout as probable success, not failure — then poll the new address to confirm.

Get it wrong and the speaker is unreachable until someone holds the reset button for ten seconds. Validate before you send: no leading zeros in an octet, no non-contiguous mask, and never the network or broadcast address.

13 — BLUETOOTH

CONTENTS ↑

setDiscoveryBluetooth:Open

WORKS

```
http://<device-ip>:8000/?Instruct=setDiscoveryBluetooth:Open
```

Make the speaker discoverable — a "pair now" button.

setDiscoveryBluetooth:Close

WORKS

```
http://<device-ip>:8000/?Instruct=setDiscoveryBluetooth:Close
```

Stop advertising.

getBluetoothName

WORKS

```
http://<device-ip>:8000/?Instruct=getBluetoothName
```

Current Bluetooth name.

modifyBluetoothName:{name}

WORKS

```
http://<device-ip>:8000/?Instruct=modifyBluetoothName:{name}
```

Rename it.

clearBluetoothPairingList

WORKS

```
http://<device-ip>:8000/?Instruct=clearBluetoothPairingList
```

Forget every paired device — useful on guest-room handover.

14 — DEVICE CONTROL

CONTENTS ↑

setPlayerCmd:devicename:{name}

WORKS

```
http://<device-ip>:8000/?Instruct=setPlayerCmd:devicename:{name}
```

Rename the speaker.

reboot WORKS

```
http://<device-ip>:8000/?Instruct=reboot
```

Restart. Expect ~60 s offline.

shutdown WORKS

```
http://<device-ip>:8000/?Instruct=shutdown
```

Power off. Requires physical access to bring it back.

factory CAVEAT

```
http://<device-ip>:8000/?Instruct=factory
```

Erases everything, including Wi-Fi credentials. The speaker drops off the network immediately, so the reply usually never arrives. It then needs setting up from scratch.

setPlayerCmd:prompttone:Enable WORKS

```
http://<device-ip>:8000/?Instruct=setPlayerCmd:prompttone:Enable
```

Button and connection sounds on.

playPromptSound:{file}:{vol} NOT IMPLEMENTED

```
http://<device-ip>:8000/?Instruct=playPromptSound:{file}:{vol}
```

The whole prompt-sound download and playback family is absent. `viewServerPromptSound` hangs for 29 seconds before failing.

15 — TRAPS WORTH KNOWING

CONTENTS ↑

Everything above, condensed. If you read one section before writing a driver, make it this one.

TRAP	WHAT TO DO
Parametric EQ JSON gets percent-encoded	Send it raw over TCP, or disable encoding on your client.
Source tokens from the manual return `FAIL`	Use the display strings from `DevFunction`.
`getplay` returns `Not`	Read `Schedule` from `getMetaInfo`.
`maximumVolume` does nothing	Enforce the ceiling in your own system.
Seek on a paused speaker looks ignored	It is queued; it lands when playback resumes.
Static IP request times out	Expected. Treat as probable success and poll the new address.
USB track numbers change	Re-read the list every time.
Spotify ignores transport commands	Expected. Volume still works.
`PlayState` says `"plays"`	Normalise it to `"play"`.
Prompt-sound commands hang	Not implemented. Do not call them.

Because control is plain HTTP on the local network, any system that can send a URL can drive an Air Mini. There is no cloud dependency, no bridge hardware and no proprietary SDK — which means these platforms need only a generic HTTP or TCP module, not a certified driver.

Control4 GENERIC TCP / HTTP	Crestron SIMPL / C# MODULE	Savant GENERIC IP PROFILE	Lutron VIA INTEGRATION SERVER
RTI INTEGRATION DESIGNER	ELAN CUSTOM IP DRIVER	URC TOTAL CONTROL IP	Home Assistant RESTFUL / COMMAND_LINE
Loxone VIRTUAL HTTP OUTPUT	KNX VIA IP GATEWAY LOGIC	Josh.ai HTTP ENDPOINT	Node-RED HTTP REQUEST NODE

Home Assistant, in full

```
rest_command:
  airmini_volume:
    url: "http://<device-ip>:8000/?Instruct=setPlayerCmd:vol:{{ level }}"
  airmini_source_usb:
    url: "http://<device-ip>:8000/?Instruct=setPlayerCmd:switchmode:USB%20Disk"

sensor:
  - platform: rest
    name: Air Mini
    resource: "http://<device-ip>:8000/?Instruct=getPlayerStatus"
    json_attributes: [PlayStatus, Volume, PlaySource]
    value_template: "{{ value_json.PlayStatus }}"
```