

SYNOPSIS

airsh :: AIR shell, a simple air-coremixer.pd control shell to demonstrate OSC parameters that control the **AIR core mixer** and **AIR console** | **B-AIR project Creative Europe 2020-2023**

This manual is formally a part of **airsh** but serves as a wider explanation of whole web audio/control framework conceived to support the B-AIR project's **AIR platform**.

PUREDATA gpp-air-coremixer ELEMENT [server-side]

OSC-controlled puredata core mixer for AIR platform:

[**gpp-air-coremixer.pd** <num.input.channels> <OSC.inaddress> <OSC.outaddress>]

Air core mixer will automatically connect to the <jackd> / <jacktrip> -provided channels via puredata input [adc~] and output [dac~]. Although technically unlimited, within this setup (AIR platform) the core mixer will provide stereo master output and a separate prelisten-bus (as an audio-premonitoring option). The number of the input channels is arbitrary, set as a creation argument to the core mixer. **gpp-air-coremix** will take the following creation arguments:

- [=] Number of channels (creation arg)
- [=] OSC receiver address and port (creation arg)
- [=] OSC sender (backlink) port (creation arg)
- [-] inlet1 : puredata [netreceive] control input (debugging purposes)
- [-] inlet2 : direct mixer input (debugging purposes)

gpp-air-coremixer and jacktrip

gpp-air-coremixer is primarily intended to be used with Stanford CCRMA **jacktrip** audio networking solution, based on Jack server, primarily in **hub mode**, following two possible audio-scheme scenarios, which differ in network-bandwidth demands, quality of user experience and flexibility:

[**full**] good networking conditions: each client broadcasting their channels and receiving all but themselves (jacktrip hubpatch mode 2). **gpp-air-console**, a separate part of **gpp-air-coremixer**, serves the purpose of monitoring and controlling this proces **[#]** and, above all, master mixing. In this scenario, each client will mix their own master and prelisten bus (via **gpp-air-console**). In this scenario, bandwidth issues may affect the very structure of mix(es).

.

[**light**] poor networking conditions: each client broadcasting their own channels but receiving only final mix (jacktrip hubpatch mode 0). Master mixer will also receive prelisten bus, but this one is mixed by master (gpp-air-coremix). A single prelisten-bus is established, available to the master-mixer but not for the other remote members. In this scenario, bandwidth issues may cause serious delay in perception of the final mix feedback.

Links, explaining the context:

[**OSC protocol**] <https://ccrma.stanford.edu/groups/osc/index.html>

[**jacktrip**] <https://ccrma.stanford.edu/software/jacktrip/>

[**jacktrip modes**]

<https://www.haven2.com/index.php/archives/jacktrip-hub-mode-server-options>

[**jack server**] <https://jackaudio.org/>

.

WHO CAN CONNECT

[**jacktrip**] Any client that can run and make use of **CCRMA Jack / jacktrip** can join. According to our pre-testing even connection via modern smartphones works decently. Jack / jacktrip is free software solution. [all open-source software]

[**gpp-air-console**] Any client that can run full installation of **puredata 0.51+** [open-source software]

[**airsh**] Any client that can run Perl programming language v5.28 or higher
<https://www.perl.org/>. Air shell is actually a simple OSC sender so it can be easily implemented in different software environments. All processing is done by **gpp-air-coremix**.

CONTROL WORKFLOW

airmix and airmon control and monitoring OSC streams

*** **client** | user action->[**airsh** or a midi mixer controller connected via **gpp-air-midisender**]

[**gpp-air-console**]->user perception | **client** ***

[-]

airsh [**air shell**], though text-based, allows for targeted and complex multi- controller operation, such as addressing complex spans of channels, describing complex ramps for controllers and triggering gadgets such as player / recorder ([#] more to come in future). This functionality (conceptually) transcends simple physical-faders-and-knobs operating mode, although, of course, will always represent 'a different kind of game'. The combination of both, external physical controller and air shell might represent a win-win combination in controlling different cases and situations during live mixing.

[-]

gpp-air-midisender is actually a simple midi frontend talking to **gpp-air-coremix** sitting at server-side. It can be set up to control any parameters, basic or high resolution faders/knobs (nrpn).

[-]

gpp-air-console is bi-directionally connected to **gpp-air-coremix** and **gpp-air-midisender** slaves to **gpp-air-console**. All the crucial init parameters such as number of channels are automatically synchronized if changed on the server. Currently, the console intended as master-mixer monitoring device, but shall be developed [#] into **client monitoring and organizing tool** in order to **provide complete mixing and input/output channels overview to each client** (in the context of 'full' audio-scheme scenario as described above).

PUREDATA **gpp-air-console** ELEMENT [client-side]

[**gpp-air-console.pd** <OSC.inaddress> <OSC.outaddress>]

[=] OSC receiver address and port (creation arg)

[=] OSC sender (backlink) port (creation arg)

[-] inlet1 and outlet1 for bi-directional communication with the optional **gpp-air-midisender** at the client side

PUREDATA **gpp-midisender** ELEMENT [client-side]

[-] inlet1 : input for **gpp-air-console** commands [connected to console's outlet1]

[-] inlet2 : midi setup - in

[-] outlet1 : number of channels and input address/port monitoring state

[-] outlet2 : **gpp-air-console** init-bang (querying server for crucial parameters such as number of channels and server-receive OSC address [connected to console's inlet1])

[~] outlet3 : midi setup - out

AIRMIX :: AIR CORE MIXER OSC COMMAND PROTOCOL

CHANNEL ADDRESSING

The structure of the OSC message is:

/airmix/<channel>/<controller> argument(s)

where channel =

master (in short: m)

prelisten | preunmute (in short: p pu)

<channel> | <channel_span> | <channel_selection_alias> - see **CHANNEL_SPAN AND CHANNEL_ALIASING**

MONITORING

<ch> mon / <ch> nom

local per-channel monitoring; provides a pd gui window for each separate channel in the selected span (server-side).

mix / xim

show whole mixer-monitor in a single pd window. mixer-monitor is a gui element directly rendered by the **server** and is intended for direct monitoring on the server side.

con / noc

inform remote client (gpp-air-console user) to show or hide console gui window (a remote version of mixer-monitor in a single pd window)

CONTROLLERS

master|m fader|f|level <airliner_vector>

send an **airliner_vector** to a master fader (both channels)

master|m lfader|rfader|left|right <airliner_vector>

send an **airliner_vector** to a master fader (separate channels)

<ch> fader|f <airliner_vector>

send an **airliner_vector** to a separate channel's faders

<ch> pan|p <airliner_vector>

send an **airliner_vector** to a separate channel pan

* **NOTE** master has no pan but both faders /stereo/ instead

AIRLINER_VECTOR

is managed by gpp-airliner~.pm. Airliner can be bound to any mixer param, currently it controls **faders** and **panning**

it will accept 1 - 4 input parameters:

<value_to_reach [rms 0-1]> <time_of_operation [ms]>? <curve [string - see below]>? <initial delay [ms]>?

where only the first parameter, **<value_to_reach [rms 0-1]>**, is compulsory. The other three, if omitted, will default to:

<time_of_operation [ms]> defaults to 'jump' micro-fader time (50 ms)

<curve [see]> defaults to 'lin' or 'linear'

<initial delay [ms]> defaults to 0

Currently available curves:

jump micro-fader (50ms) linear curve

lin linear

sin sinusoidal (soft)

hsin half-sinusoidal (equal power)

log logarithmic, hard neck

pow power (exponential), slow sudden attack

<airliner_vector> examples

0.9 => jump to 0.9

0.422 7500 => slide to 0.422 in 7.5 seconds

1 4230 sin => slide to 1 in 4.23 seconds following sinusoidal curve

0.5 7600 pow 2000 => slide to .5 in 7.6 seconds following power curve, but start after 2 seconds from now

AUDIO PRODUCTION / PROCESSING TOOLS

PLAY

<ch> play <file>

Play a file (wav,aiff)

<ch> play <file> <hh:mm:ss.uuu>

Play a **file** from the selected **timing** on

<ch> play <file> <hh:mm:ss.uuu> <hh:mm:ss.uuu>

Play a **file** from the selected **timing** to the selected **end**

<ch> play <file> <hh:mm:ss.uuu> <hh:mm:ss.uuu> +<hh:mm:ss.uuu>

Play a **file** from the selected **timing** to the selected **end** with the selected **+delay**. Note that the player / looper delay parameter should be marked with +

<ch> stop

Stop current playing session at channel(s) <ch>

LOOP

<ch> loop <file>

Loop whole **file**

<ch> loop <file> <hh:mm:ss.uuu> <hh:mm:ss.uuu>

Loop **file clip** from selected **timing** to selected **timing**

<ch> loop <file> <hh:mm:ss.uuu> <hh:mm:ss.uuu> +<hh:mm:ss.uuu>

Loop **file clip** from selected **timing** to selected **timing** with intermediate **+delay**

<ch> stop

Stop current looping session at channel(s) <ch>

RECORD

<ch> record <file>

Record to **file**

<ch> record <file> <hh:mm:ss.uuu>

Record to **file** after initial **+delay**

<ch> record <file> <hh:mm:ss.uuu> <hh:mm:ss.uuu>

Record to **file** after initial **lead-in** till selected **end**. The second figure being absolute time, not a time interval. Timing display will change color to red when the recording actually starts.

<ch> rstop

Stop current recording session at the addressed channel

OTHER

gpp-playhed will be added in future to enable complex manipulations of PLAY function (speed change, different ways of oscillating ramps, granulation/heartbeat etc...) [#] Performance-sensitive, will have to be tested in terms of mass channel addressing and potentially heavy monitoring OSC-traffic.

CHANNEL_SPAN AND CHANNEL_ALIASING

<ch> refers to:

channel number : send to a selected channel

all : send to all channels

even : send to even-numbered channels

odd : send to odd-numbered channels

selection span : send to a selection span which consist from '-' delimited ranges and ',' delimited lists, for instance **1,3,5-7**

GLOBAL AND PER-CHANNEL COMMANDS

solo, mute

<ch> **solo** : set the channel 'solo'

master mute | **m** : mute master channel (monitored via setmute boolean)

<ch> **mute** | **m** : mute selected channel (processed and monitored via **setmute** boolean)

master unmute | **u** | **um** : mute the master channel (monitored via setmute boolean)

master lunmute | **lu** : mute left master channel (monitored via **setlmute** boolean)

master runmute | **ru** : mute right master channel (monitored via **setrmute** boolean)

<ch> **unmute** | **u** : mute selected channel (monitored via setmute boolean)

<ch> **setmute** <boolean> : set mute value (monitoring: /airmon/etc/<ch>/setmute <boolean>)

master setlmute <boolean> : set l-master mute value (monitoring: /airmon/etc/master/setlmute <boolean>)

master setrmute <boolean> : set r-master mute value (monitoring: /airmon/etc/master/setrmute <boolean>)

NOTE: solo also operates on complex spans, for instance 1-3,7 will set solo the corresponding channels.

prelisten bus

ch prelisten | **pl**: add the channel to the pre-listening bus (separate prelisten monitoring audio port)

ch premute | **pm**: remove the channel from the pre-listening bus (separate prelisten monitoring audio

port)

ch presolo | **ps**: set the channel as pre-listening **solo**

ch setpremute : boolean directive (fit for monitoring: /airmon/etc/<ch>/setpremute <boolean>)

other globals

panic : all channel faders set to 0

dir | **cd** : set global wavedata directory

ls : get global wavedata directory

trimfader : set **trimfader time** or **trimfader type** [#]

centerpan : center all pans

fullout : set all faders to full

stereopan : set even-numbered faders to 0 and odd-numbered faders to 1 (receiving in full stereo mode)

globepan : evenly distribute panning across panorama 0-1 accross all channels

setff : set server's fanning factor (distance between serial OSC monitoring request, somewhere between 2 and 6, default 3) :: the serial monitoring requests (fader, pan) get distributed after the following formula:

setmode : set server's monitoring mode (auto, mirror, coarse). See 'Two main server-side monitoring modes' below.

fader : (<ch> * <FF> * 2) msec DELAY

pan : (<ch> * <FF>) msec DELAY

INTERNAL CLIENT- COMMANDS

These are client- (console) initiated

/airmix/cinit/ bang => client signaling server their own loadbang and querying for initial parameters such as number of channels, airdir etc. This request is handled separately on air-coremixer input.

PRELISTEN BUS

Unless in the **light** audio-scheme scenario **Prelisten bus** itself will not be provided by the server. It will be rendered by the client, gpp-air-console will render their own bus from all channels gathered via jacktrip.

On the contrary, **light** mode [#] will provide prelisten bus, as well as master-mix.

ch prelisten preunmute | **pl pu**: add the channel to the pre-listening bus (separate prelisten monitoring audio port) (monitoring: /airmon/etc/prelisten 1)

ch premute | **pm**: remove the channel from the pre-listening bus (separate prelisten monitoring audio port) (monitoring: /airmon/etc/prelisten 0)

ch presolo | **ps**: set the channel as pre-listening **solo** (monitoring: /airmon/etc/presolo)

setprelevel : set the level of prelistening bus. Bound to 'jump' (50ms) ramp, not gpp-airliner. This know is per-ch only. There is no master volume. Available via **/airmon/etc/[ch]/setprelevel**

AIRMIX EXAMPLES [broadcast by the user/client sender such as airsh]

MASTER

/airmix/master/lfader 1 : set left master fader

/airmix/master/rfader 0.235 : set right master fader

/airmix/master/fader 0.5 : set both master faders

CHANNELS

/airmix/3/fader 1 : set fader #3
 /airmix/3-5,7/fader 1 : set faders #3,#4,#5 and #7
 /airmix/odd/fader 1 : set odd faders
 /airmix/all/fader 1 3000 sin : start all faders' ramp to 1, 3000 ms, sinusoidal curve
 /airmix/even/pan .5 6000 pow : start even pans' ramp to .5, 6 sec, power curve
 /airmix/4,5/pan 0 4000 log 6234 : start channel's 4 and 5 pan ramp to 0 in 4 s, logarithmic curve, after 6,234 sec delay
 /airmix/even/play test.wav : play file test.wav on even channels
 /airmix/even/play test.wav 0:05 : play file test.wav on the even channels from 0:05 to the end
 /airmix/even/play test.wav 0:05 : play file test.wav on the even channels from 0:05 to the end
 /airmix/1-4/loop test.wav 0:20 0:22.123 +0:03.120 : loop file test.wav on channels 1-4 clipped at 0:20 to 0:22.123 with 0:03.120 delay/gap
 /airmix/3/record myrecording.wav 0:25 : record to myrecording.wav after 0:25 delay (with pre-count)

CHANNEL- AND GLOBAL- ALIASES

/airmix/odd mute : mute odd channels
 /airmix/3,4-5 solo : solo channels 3,4 and 5
 /airmix/panic : panic (turn all faders down)
 /airmix/ls : get info about the server audio storage directory
 /airmix/trimfader : set trimfader time and type
 /airmix/centerpan : center all pans
 /airmix/stereopan : set all pans to stereo mode
 /airmix/globepan : equidistant placing of all pans
 /airmux/setmode : set server monitoring mode
 (...)

AIRMON :: AIR CORE MONITORING OSC COMMAND PROTOCOL

MONITORING TACTICS

Serves dispatching information about the mixer state to all subordinated (remote) web client subjects. While the data, dispatched on momentary basis, does not represent a problem regarding the OSC traffic demands, all continuous data types, such as fader pan etc, represent a challenge. Air engine will internally handle such types by the usage of signal-level streaming, therefore monitoring can be done using 3 different tactics:

A request : whole **request** (for instance: fader .5 3000 sin 6000) is transferred (copied from the airmix OSC input) to the web clients and clients themselves will take care about rendering itself, from beginning to end. Not only **gpp-airliner** directive, but also timer requests can be given this way.

B snapshot : current state of controller transferred on periodical basis (metronome pulse). Intermediate request data is sent using relatively low frequency in order to avoid OSC wire overheads

C contour : this tactics is actually a modification of **snapshot** tactics, but will only send **snapshot** to the client(s) when **RF (request fulfilled)** signal on the respected controller is broadcast (this is called **VR (value reached)**, also provided by gpp-airliner~). So instead of metronome pulse, the initiative for broadcasting **snapshot** is based on reaching controllers' border states. All intermediate request data is ignored. This tactics alone can be used for low bandwidth cases, but also to complement tactics **A** and **B**. With the **A** tactics, it can help assure that the final state is obeyed regardless of potential server/client desync. With the **B** tactics it can assure that the border state information is delivered exactly in time, even if the server's snapshot heartbeat is low.

OSC CONTROLLER MAP

MONITORING TACTICS:

A. REQUEST

/airmon/request/<ch>/<controller> <request params>

any kind of user-request as such, copied directly from the OSC input for each respective controller, such as:

/airmon/request/<ch>/**fader** <airliner~ params>

/airmon/request/<ch>/**pan** <airliner~ params>

—

A2. VIRTUAL REQUEST

A request assembled by the server engine (not passed by the user) in order to encapsule bandwidth-demanding movement to be rendered by the client engine, such as:

/airmon/request/<ch>/**timer**/<type> [start|stop] [initial_timing_ms]?

display timer, where <type> can be "play" or "record"

—

B. SNAPSHOT

/airmon/snapshot/<ch>/<controller>

controller state at the very moment

C. CONTOUR

/airmon/contour/<ch>/...

separate reporting of basic border states :: the point of this tactics is to report some basic params that the client (web or pd) can use to render their own movement. There are several 'contour handlers', such as:

... **rf**/<controller> => **controller** rf when reached (request fulfilled => reached destination boolean (1=yes 0=not yet), fader or pan

... **vr**/<controller> => **controller** vr when rf reached (value reached => the actual destination value triggered when reached, fader or pan, and also play, record (border states)

... **ac**/<controller> => **controller** ac (active) : play, record and such things => player is active boolean (1=yes, 0=not)

*note difference between **rf** and **ac** switches: **rf** policy is negative (= positive when not active) while **ac** policy is positive (= negative when not active).

ALL THE REST

All data that cannot be ranged into the 3 main monitoring tactics, will be passed via **etc**

/airmon/etc/<anyparam> => other data or switches, non policy-classified

/airmon/etc/<ch>/<anyparam> => other data or switches, non policy-classified : per-channel

for instance:

/airmon/etc/<ch>/filename/ => current file name distributed exactly before each play- or record-request

/airmon/etc/<ch>/mute => mute boolean for the selected channel

/airmon/etc/master/mute => master mute boolean (both channels)

/airmon/etc/master/rmute => master mute boolean (right channel only)

/airmon/etc/master/lmute => master mute boolean (left channel only)

DIRECT SYS PARAMS

System params (internal communication between air-coremix and console, without user interaction) are sent via /sys/ and /syscore/ controllers. **/sys/** will deliver the message to the console frontend (shell that envelopes console-core) and **/syscore/** will to the console core itself. These params is no thing the end user should be bothered with.

/airmon/sys/CHNUM/ => number of channels, required by gpp-air-coremixer creation argument - the console automatically resizes if the argument changes.

/airmon/syscore/AIRDIR/ => airdir path storage (path itself stored as text)

/airmon/syscore/SHOW/ [boolean] => show / hide console

TWO MAIN SERVER-SIDE MONITORING MODES BASED ON DIFFERENT MONITORING TACTICS

MIRROR and **COARSE** :: implementing **contour** and **snapshot** tactics:

MIRROR : use **both** tactics => high bandwidth and low client capacity; will report border states and intermediate snapshots according to network capacity (tunable). **Contour** skeleton is used to prevent possible timing-based anomalies, especially with extremely low frequency monitoring heartbeat.

COARSE : use **only contour tactic** => low bandwidth and low client capacity; will only report border (crucial) states.

AUTO :: use **request** and **snapshot** tactics => client themselves will render events; this is only safe when client is capable of running puredata console. Suitable for very low web bandwidth. **Contour** skeleton is used to prevent possible timing-based anomalies.

Server can be set to either mode in terms of communicating with its console(s):

/airmix/**setmode mirror** or airsh **setmode mirror** :: set mirroring mode (contour + snapshot)

/airmix/**setmode coarse** or airsh **setmode coarse** :: set mirroring mode with only contour tactics enabled

/airmix/**setmode auto** or airsh **setmode auto** :: set autonomous mode (all rendering based on request tactics data vectors)

/airmix/**lsmode** :: list currently selected monitoring mode

AIRSH EXAMPLES EXAMPLES> [broadcast by user/client]

airsh m fader 1 2000 sin => fade complex curve

airsh 1 fader .3 4000 log 2000 => fade after delay

airsh play file.wav 0:12.345 0:15.012 +1:0.300 => play a clip after delay

airsh loop file.wav 0:12.345 0:15.012 +1:0.300 => loop a clip after delay

airsh rec 0:10 0:15 => record from (abs) time 10 to 15 seconds (5 sec long block)

AIRMON PROTOCOL EXAMPLES [broadcast by gpp-air-coremix/server]

/airmon/request/master/lfader/ 1 3000 sin

/airmon/request/master/rfader/ 1 3000 sin

/airmon/request/master/fader/ 1 3000 sin

/airmon/snapshot/1/fader .346567

/airmon/contour/master/rf/lfader 1

/airmon/contour/master/rf/lfader 1

/airmon/contour/3/ac/player 1

/airmon/contour/1/timer/player start 3543 down

/airmon/contour/1/timer/player stop

/airmon/contour/3/timer/recorder start 0 up

AUTHOR

Gregor Pirs <gregor.pirs@guest.arnes.si>

VERSION

[27-12-2022]