SYNOPSIS

airsh :: AIR shell, a simple shell to control gpp-air-coremix.pd via OSC

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

DISCLAIMER

This software is based on author's collection of gpp- patches and perl- based tools and has been developed within the frame of the **B-AIR project Creative Europe 2020-2023**. The project has been funded with support from the European Commission. This document content reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

HOW TO RUN airsh itself

airsh with no arguments : connect to localhost, no init script

airsh host init-script-path: connect to **host** and run **initscript**. In this case, host should be declared, if the server is running locally, use "localhost"

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

OSC-controlled puredata core mixer for AIR platform:

[**gpp-air-coremix.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-coremix and jacktrip

gpp-air-coremix 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 air mixing system, 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**] poorer networking conditions: each client broadcasting their own channels and receiving minus-mix (all but theirselves) from the air mixer at the server. The Master Mixer receives master and prelisten outputs (2x stereo). In this scenario, bandwidth issues may cause latency issues / synchronization issues of the final mix and its feedback. In practice, this (light) option has been proved very successful and reliable and also simpler to manage and configure at the server.

Links, explaining the context:

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

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

[**jacktrip**] `https://ccrma.stanford.edu/groups/soundwire/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**.

.

CONNECTIVITY

A channel input can be connected either MONO or STEREO. Currently, there's no option to recognize whether the jack has been mono or stereo connected within PD, so only manual option remains:

<ch> mono : set selected channels into mono mode

<ch> stereo : set selected channels into stereo mode

mono : shortcut to set mono mode globally

stereo : shortcut to set stereo mode globally

IMPORTANT A **mono** wire should be connected to **both stereo inputs** in order to assure proper operation.

A stereo/mono led indicator on the top of each channel fader shows the stereo/mono state.

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** / **gpp-air-iconsole** 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. The i-console version is an interactive version of the console and provides mouse-mixing gui (via Puredata), developed to enhance awareness of the non/synchronicity between the action-herenow and the remote result. The gpp-air-coremix server gui (if enabled) also follows the same (interactive) model, where the action and effect are represented with separate controllers.

.

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 addres [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 **airliner_vector** to a master fader (both channels)

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

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

<ch> fader|f <airliner_vector>

send **airliner_ector** to a separate channel's faders

* faders are dynamic. a ramp can be interrupted with a new ramp at any point and will continue to flow from that point on. Airliner vector curves will **scale** up/down to adapt to the height of the new window to cover.

<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 **+**

<file*> can be either a **filename in connection with current working directory** (see cd / dir directives) or an **absolute path**

<ch> wplay <same arguments as play>

play a **file** with waitmute enabled. Initially, the recording is muted while being played and it will unmute the moment that the wait-unmute condition is met (=a moment of silence). therefore, the apperance of the sonic matter will be 'organic' = not forefully cut.

set wmtime <milisecs>

set wmtime - required time of relative silence between sound events to trigger waitmute / waitunmute.

set wmthreshold <-dB>

threshold of recognition what means 'relative silence' using -dB scale (-100 to 0 dB).

<ch> stop

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

<ch> wstop

stop the **file** with waitmute enabled. The **stop** directive will wait till the wait-unmute condition is met. so the stopping will look organic, but may not happen at the very moment of execution of the command.

<ch> wclip <same arguments as wplay / play> [**WARNING: STILL FLIMSY!!]

Auto clipper, based on waitmute system.

wclip uses the same timing parameters as **play** command. The timings are rather arbitrary, though, and determine only the timing-frame within which the waitmute system will operate (= initial wplay start splaying at the timing1 and the timing2 will end playing anyway, even if the waitmute condition hasn't been met yet).

set wctime <milisecs>

set wctime - required minimal duration of non-silence segment [> wmthreshold dB] to recognize the clip existence and trigger the wstop chain.

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> immediately

<ch> **deloop**

stop playing when the current loop iteration finishes

<ch> **unwind**

release the loop and **play the sample till the end**

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** a **lenght** long clip. The second figure being interval length. Timing display will change color to red when the recording actually starts.

<ch> stop

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' : set corresponding mutes and also send **setsolo** boolean signal

<ch> unsolo | **us** : set the channel 'not solo' : set corresponding mutes and also send **setsolo** boolean signal

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>)

mutelock directive is machine-generated and controls the second mute layer during solo operations

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

solo and presolo switches policy

solo and **presolo** switches do the following:

solo or a group of solo commands (represented by a **complex channel span** will grab the mixer, disabling the rest of channels. If these are already muted, they will stay muted. If not, they will be **solomuted** (= mechanically locked to mute-state). If yes, they will remain muted, but an attempt to unmute them will put them into **solomuted** state.

solo-ed channels can be muted/unmuted at will

Any channel can be either added to or removed from the **solo span**.

When the last **solo** channel is disabled, all **solomuted** channels will **return to the previous state**, either **muted** or **unmuted**.

same policy applies on **prelisten** bus solo (**presolo**)

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** : adjusts all **premute** values and also sends **setpresolo** boolean to the corresponding channel(s)

ch unpresolo | **ups**: set the channel as pre-listening **solo** : adjusts all **premute** values and also sends **setpresolo** boolean to the corresponding channel(s)

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

ch setsolopremute : machine generated solomute directive on presolo bus

waitmute | wm, waitunmute | wu

waitmute or **wm** performs an ordinary <mute> command, but before it does it waits for a condition to be fulfilled: the post-fader (and pre-mute) output of the track (or both tracks of a stereo) in question will have to become 'silent', with env~ value < 10.

waitunmute or **wu** performs an ordinary <unmute> command, but under the same condition than **waitmute**

NOTE if the post-fader output level of the track is below the env~ treshold (< 10) **waitmute** and **waitunmute** function similarly to **mute** and **unmute**

other globals

human | **nonhuman** : set OSC input policy. **nonhuman** will: 1) not dereference complex spans (only per-channel or global addressing remains) and 2) treat all timing arguments as miliseconds (not hh:mm:ss.msc format). **human** reverts this behaviour to normal (enabled complex spans and hh:mm:ss.msc format)

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

lstracks : list numbers / configuration of input and output tracks through the rightmost outlet of [gpp-air-coremix] element (via set message)

autosave : turn autosave (track names, mixer state, fader type) feature ON

noautosave : turn autosave (track names, mixer state, fader type) feature OFF

latency / **latencystop** : starts / stops measuring console latency (audio and control). The values are reported to the control (3rd) outlet of the [gpp-air-coremix] element (via set message)

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

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:

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

alivesig start / stop : turn alive signals on / off. In the left bottom corner of the iConsole and Coremix there are rectangular symbols **iCo** and **Srv**. If **alivesig** is running they will show the state of connection:

> **iCo** is green : iConsole is connected (server-side monitoring)
>
> **Srv** is green : Server is connected (client-side monitoring)
>
> both red in case of opposite

reset : erase autosave mixer data. By default, mixer has 2 sec interval autosave (autosave.txt). Persistence survives mixer-restart and mixer-resize. This command resets all parameters.

fadertype lin / log : set linear or logarhytmic fader tipe. **setfadertype** boolean [NOT YET IMPLEMENTED ON iCONSOLE!!!]

INTERNAL CLIENT- COMMANDS

These are client- (console) initiated

/airmix/cinit/ bang => client signalizing server their own loadbang and querying for initial parameters such as number of channels, airdir etc. This request is handled separately on air-coremix 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, logarhytmic 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 transfered (copied from the airmix OSC input) to the web clients and clients themselves will take care about rendering itself, from begining 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 gpp-air-coremix and gpp-air-(i)console, without user interaction) are sent via /sys/ and /syscore/ controllers. **/sys/** will deliver the messaga 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-coremix 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

OFF :: monitoring disabled

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 lo 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

AIR MIXER CONFIG FILES

/var/air/data/mixer/airdir.txt - air current working directory

/var/air/data/mixer/state/autosave.txt - mixer params state

/var/air/data/mixer/state/autosave_chname.txt - fader type and channel names

/var/air/data/airsh/airsh_log.txt - list of recently cast airsh commands

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 15 seconds after 10 seconds delay

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

[27-2-2024] | gpp-air-coremix.pd 1.6