WING

PERSONAL MIXING CONSOLE

WING Remote Protocols

OSC remote control, MIDI SYSEX, Binary Interfaces,
and `wapi`, an API for WING

[V3.1.0-2 - Wing FW 3.1.0 and above]

Table of Contents

Introduction

Introduction

About this document

My name is Patrick-Gilles Maillot and I am authorized by Behringer to publish and maintain this “WING remote protocols” document; I am not a MusicTribe employee.

Starting with release 2.0 of the WING firmware, OSC and native remote protocols form a single (this) document under two separate sections, and share the same series of appendix chapters.

Most users will probably find it easier to remote access their WING with `OSC` commands while more advance programing and less restricted control are possible using **native** commands and the Wing API (**wapi**) library.

While the main purpose of this document is to offer developers with a reference to programing their WING console using `OSC` or `wapi`, this document also includes chapters which would qualify more as User Guide oriented ones, helping novice and advanced users to better use the desk: Dedicated chapters provide additional details on `MIDI`, `Custom Controls`, `Effects`, `Plugins`, `Shows` and `Scenes`, `Routing` and more.

I want to thank the Behringer development team for their continuous support in writing this document.

General features of the WING console

In 2019, Behringer has been designing a whole new digital mixing desk they would later call “Personal Mixing Console”. The WING was unveiled to the public in November 2019 and first shipments took place in December that year. As to why calling it a “Personal Mixing Console”, here is a perfectly valid answer from one of the fathers of the console: “A fundamental idea of WING was providing a high level of customization options to the engineer, allowing to adapt the console surface to his personal preferences and needs”.
The WING console was awaited by several X32 and M32 users as it carried the promise of new features, long expected since the first release the X32 and M32 family of digital mixing desks. It seems the WING receives a warm welcome from the community.

The Behringer WING provides 48-channel, 28-bus mixing with 24 motorized faders and a large 10” capacitive-touch LED screen. The desk is designed for live performance, live and studio recording, touring sound, A/V, club installs, and more. Three separate fader sections and a custom controls section can be easily and intuitively tailored to personal requirements.

The 48-channel inputs [in/aux] and 28-channel mixes [bus/matrix/main] can all be in mono/stereo or mid-side mode, with specific source mutes and metering, and provide dynamics, EQ and FX processing. They too can be given a color, icon, name and up to 8 console or user defined tags for grouping and filtering purposes.
WING input channels provide low-cut & high-cut filters, tilt-EQs, all-pass or Sound Maxer, in addition to a 6-band parametric EQ. All buses, matrices, and mains feature 8-band parametric EQ. All channels and buses can also load high-end simulations modeled from hardware devices such as Pultec EQ, SSL Bus Compressor and Gate/Expander, SPL Transient Designer, Neve EQ, Compressor and Gate, Focusrite ISA and D3, DBX160,

LA-2A, 1176, Elysia mPressor, Empirical Labs Distressor, and more. The built in FX rack supports 8 true stereo processors including TC VSS3 algorithms, Lexicon, Quantec, and EMT emulations. Other processing includes modulation, equalization, dynamics, nonlinear effects and four guitar amplifiers with cabinet simulations. A maximum of 16 stereo inserts can be used for applying internal FX or outboard processing to input channels or buses.
The channel editing section provides instant channel status overview and flow of operation. It allows working on the selected channel processing, even when the main display is used for something completely unrelated. Touch-sensitive rotary controls allow you to display the most relevant information, all at your fingertips.
The central Custom Controls section[1] offers user-assignable controls including 4 rotary encoders and 20 buttons with 2 LCDs that can be set as functions readily available.
A big rotary wheel offers fine-adjustments for up to 8 user parameters or can be used for DAW remote control via USB MIDI.
The control configuration also includes predefined functionality for USB and SD-card recorder transport, show control and mute groups.
WING includes 8 (full size console) or 24 (Rack and Compact consoles) original MIDAS PRO microphone preamps and 8 XLR outputs with professional quality specifications. 8 TRS line auxiliary ins and outs help bring in signals from media players or computers.

A brand new StageCONNECT[2] interface allows connecting breakout boxes and delivers up to 32 channels of low-latency input or output over a single standard XLR cable (DMX).
WING can accommodate 376 inputs[3] and 374 outputs thanks to 3 AES50 SuperMAC audio networking ports, which connect to digital stageboxes. In addition, 144 input and 144 output streams can be shared with other mixing consoles.
There are 48 channels of USB audio and 64 channels of Audio over IP (AoIP module optional), plus AES/EBU stereo I/O. The WING expansion card slot features the LIVE SD recording card with 64x64 channels of audio or can accommodate option cards for various standards such as ADAT, MADI, DANTE, and WSG.
All digital processing takes place on 40-bit floating point Digital Signal Processors, at 48 or 44.1 kHz, with a 1.3ms round-trip latency[4].
WING provides MIDI In/Out and 2x2 GPIO (General Purpose Input Output, 1x2 on the compact console) that can be used as console event triggers and external show controls, including power-on delays for external gear that needs sync powering with the console.
Automixing is also implemented, with 2 groups of gain sharing on any 16 input channels. The management of the respective input channel gains depends on the levels received, reducing the sum gain in the group to maintain intelligibility and low noise during meetings, ideal when several speakers are collaborating to corporate events, panels, broadcast applications or house of worship.

[1] Not on all WING family devices, the provided description matches the full-size console, unless mentioned otherwise
[2] https://www.klarkteknik.com/series.html?category=R-KLARKTEKNIK-STAGECONNECTSERIES
[3] Not considering User Signal/Patch, FX send, or Bus send entries which overlap with actual sources and would bring this to a virtual value of 478 on the WING
[4] Typical value for Out-to-In trip without effect or insert.

Wing, a family

Starting with FW version 3.0 in fall 2024, WING firmware addresses more devices than the 'big' WING which has been on the market for almost five years. Note these will be disjoint firmware packages.

Compact and Rack are now available and offer the same overall functionality than the initial WING, although the change in format implies modifications, some more important than others. We list below the main/obvious elements of WING, Rack and Compact.

WING:

- Large control surface with a Left bank of 12 channel strips, a Center bank of 8 channel strips, a n area with CC encoders and buttons, and a Right Bank of 4 channel strips
  Each bank comes with a specific group select set of buttons (IN, AUX, MATRIX, MAIN, BUS, DCA, USER1, USER2, …) on its left side
- Available in gray or black
- Each channel strip is stereo and composed of a motorized fader, a MUTE, SOLO and SELECT button, a stereo vu-meter with clip, dynamics, and gate indicators, and a 2-line B/W scribble with a color LED above it
- Big Wheel, 4 cursor Buttons, and a DAW control button for enabling some of the fader banks to control MIDI connected DAWs
- A secondary screen besides the main LCD to control/manage 'in channel' settings such as EQ, COMP, etc.
- 8 local Inputs, 8 Aux inputs, 8 aux outputs, 8 local outputs
- 2 headphones jacks on the consoled rear surface sides (left and right), level control is on the top surface
- 4-pin, 12V XLR lamp socket on the rear of the console
- Main board with Power, 2 Ethernet sockets, a USB port, 3 AES50 ports, 1 StageCONNECT port, AES-BU In and Out ports and an expansion slot fitted with 2 SD recording/playback expansion card.
- 4 GPIO (2 TRS)
- MIDI IN and OUT
- A dedicated section for USB, Monitoring
- A large CC section with 2 sets of 8 buttons for CC, transport controls, and other WING dedicated functions (such as Mute Groups), DAW control and two LCD for displaying active functions
- A dedicated section with 4 additional encoders and 4 buttons and LCD, and 8 buttons to select functions

Rack:

- 19", 4U rack format
- No wheel, no DAW control button(s)
- DAW Mode on Rack is limited to CC buttons and one Mackie Control device.
- 24 Local Inputs (8 on the WING)
- No local Aux IO (8 IN, 8 OUT on the WING)
- 4 headphones outs mapped to the 8 local OUTs at the rear of the console with their own headphone amps for IEM applications
- 1 general headphone out with dedicated level knob on the front
- No faders, and a limited set of buttons on the surface.

- No encoders below the LCD touch screen (controls are only via the touch screen, and two encoders on the right side)
- 1 LCD + set of 4 encoders and 8 buttons CC section, with Layer level indicator, VIEW and >4 <4 buttons
- 4 GPIO (2 TRS)
- MIDI IN and OUT
- No Lamp socket
- A USB socket, and 4 buttons dedicated to selecting INPUT/AUX, BUSES/MAIN, DCA/Mute Groups, and CUST TRANSP (USB, SD1, SD2) on the CC section above
- Same Main board as on WING, no fan

Compact:

- Smaller (19" rack compatible) surface with 3 sets of 4 faders and 1 main fader, with bank layer selection available from a vertical set of buttons on the left side of the console.
- 4-pin, 12V XLR lamp socket on the top surface of the console
- Similar channel strips as on WING
- No wheel, no DAW control button(s). Control is possible from the LCD screen.
- 24 Local Inputs (8 on the WING)
- No local Aux IO (8 IN, 8 OUT on the WING)
- Monitor/USB section on the right side of the screen
- 16 buttons in two vertical sets of 8 with a dedicated LCD acts as a CC section between the 12 faders left and the main fader
- 2 GPIO (1TRS) on the rear of the console (4 GPIO / 2 TRS on the WING)
- MIDI IN and OUT
- 1 headphone lack on the rear of the console (level control is on the top surface)
- Same Main board as on WING

Sources vs. Inputs

Unlike many digital or analogue desks, WING makes a clear separation between Sources and Input channels; Historically, consoles focus on input numbers assigned to Channels and Auxes. WING is offering a different perspective by focusing on the Source as the reason for any mixing. Sources can be in mono, stereo, or mid-side mode, own headamp parameters like gain and phantom power, with specific source mute and metering. They can be given a color, icon, name and up to 8 console or user defined tags for grouping and filtering purposes. All of this describes the actual Source first, before being patched to Input channels which focus on processing or mixing.

This patching process (also called “Routing”) is described in a specific “user-guide” like Appendix later in this document to help new users grasp the basic operations involved in assigning a physical source (local or remote) to a channel for mixing, as well as assigning WING processed audio data to a physical output (local or remote).

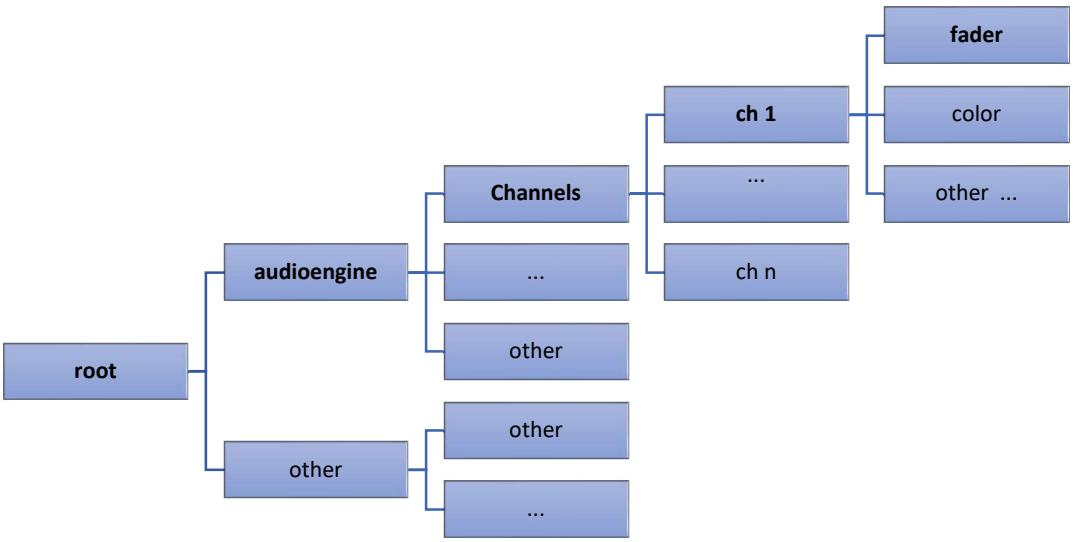
Sources can be labeled using the WING Co-Pilot app or other means such as **OSC** protocol described later in this document or the **wapi** function calls also presenter in the upcoming chapters[5], and no matter if the signal is patched to a channel, to SD recording or to any other output, it can always be referred to as its assigned Source label.

Notes
The internal real-time clock (RTC) is powered by a super-capacitor. If the WING is off mains for more about two weeks, it will most likely lose its clock data.

[5] Refer also to https://github.com/pmaillot/wapi

WING Internal Data

Like all digital or programmable devices, WING relies on an internal set of parameters that are stored/saved in non-volatile memory. This enables you to find the console in the same state you left it when powering it OFF. WING data set is very large, and in line with the many features the console offers. Each button, each attribute, color setting, effect, parameter, etc. can be found as an internal variable, member of a hierarchical tree structure.
The WING tree is more than 25000 elements! To organize this large set of internal variables, WING uses a hierarchical tree of data, stating with a root and dispatching parameters into logical groups (sub-trees or branches) until the last element (leaves) that represent the actual parameter.
For example, the `fader` associated to `channel 1` is part of the `channels` sub-tree, and is one of the many attributes of channel 1. The channel sub-tree is part of the `audio-engine`, itself at the `root` level.
A quick representation would be as shown below:

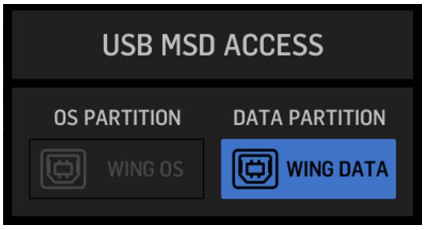


Computers use specific data structures to represent trees. WING uses one of them, based on `JSON`[6] notation. It is important to know/understand the list of sub-trees (nodes), and leaves (parameters) WING contains as this is how you can access to data. More detail on the WING data set is provided in appendix.

WING File System

At the difference of the X32, WING can be directly connected to a computer via USB; There are two ways WING can be visible to your computer, depending on the setting of the `SETUP→GENERAL` screen (shown below, with WING connected as an active data partition):

[6] JavaScript Object Notation: an efficient way to represent structured objects. Also used as a data-interchange format.



When actively connected to your PC either as an OS partition or a Data partition, the status at the top of the WING screen will show a red OS or DATA tag.

OS partition

WING can be seen as an `OS PARTITION`, a directory where you can deposit the FW release you will use to boot from at next power up or reboot. Use with caution!

Data Partition

A USB connected WING presents itself as an external disk drive. Therefore, the standard cautions apply when connecting and more important, disconnecting from the computer; ***Ensure you unmount the WING file system to avoid losing data.***

If the choice for `USB MSD ACCESS` in `SETUP→GENERAL` is set to `DATA PARTITION`, the WING file system will show as a standard external USB drive. There may be some folders already there, such as 'global' or 'shows', with subfolders, such as 'global/ch_presets', 'global/fx_presets', 'global/routing_presets', 'global/snapshots'.

Console Shutdown

How to ensure proper shutdown of the console? How to avoid data loss at shutdown?
These are FAQs. The answer is simple: Follow the instructions on the screen after going to `SETUP` and selecting the `SHUTDOWN` option. This will ensure the last changes made to the console will be saved before shutdown, and recalled when switching the console back on again.
Make sure you disconnect the outputs of the console or switch off output devices before switching off the console to avoid loud pop noise sent to connected output devices.
Nevertheless, the console FW saves its current state every `120s` if changes occurred. A command[7] enables instantaneous save of the current state before powering off the console if using remote/network commands.

[7] See /$ctl/$globals/$savenow further in this document

Remote communications with WING

WING communicates via ports 2223 [UDP], and 2222 [UDP, TCP];
Initiating a communication with WING starts with sending the 5 bytes [UDP] datagram 'WING?' to the IP of your WING, port 2222.

WING will reply to the requesting IP and port with the following datagram:

'WING,' [c_ip] ',' [c_name] ',' [c_model] ',' [c_serial] ',' [firmware]
Where:
[c_ip] e.g., '192.168.1.62'
[c_name] ascii characters
[c_model] 'ngc-full' (standard Wing console)
[c_serial] serial number (ascii)
[firmware] version string (ascii)

For its native communication format, WING proposes 14 'communication channels' to enable separation between the different 'engines' or main blocks of the console. The following communication channels are currently in use:
Control-engine (a TCP communication channel) is using channel #1
Audio-engine (main TCP communication channel) is using channel #2
Meters (UDP communication) are using channel #3

OSC uses a single UDP communication port: 2223

Keeping connections alive

Open connections will time out after 10 seconds of inactivity (on the receiving side). One way to keep a connection active is to request at regular intervals of less than 10 seconds some data from the console. There are many data that can be collected, as shown later in this document.

Number of simultaneously connected applications

WING can simultaneously communicate with up to 24 connected 'clients'; Make sure your network bandwidth supports this, or limit your number of simultaneously connected clients to a lower value; The console will reject connection requests if the maximum number of simultaneous connections (24) is reached.
What we call 'clients' above refer to actual TCP ports that communicate with the console. Some applications may use several ports and this will reduce the actual number of applications that can simultaneously connect and communicate with WING.
UDP communications such as used for OSC do not have this limitation, being "connection-less". On the other hand, WING's OSC remote protocol enables only one (1) subscription to data (for receiving event messages) at any given time.

Subscriptions must be kept alive; they automatically die after 10 seconds.

Accessing WING Internal Data and Functions from remote programs

As mention in the introduction, WING hosts an **OSC** compliant remote protocol server that offers access to the full set of features of the desk. This is described in the "**WING OSC protocol data interface**" chapter below.
WING also offers a native, binary protocol with the capability to access (read or write) parameters of its internal structures and take full advantage of the entire set of features of the digital desk, including remote control. The protocol is fully described in the "**WING native/binary data interface**" chapter below.
To help users access the native protocol, a WING API written in C [**wapi**] has been developed and is available as a free resource at https://x32ram.com/wapi to write programs that directly call **wapi** functions.

WING OSC protocol data interface

OSC Remote Protocol

WING includes an OSC Remote Protocol server. This enables easy access to remote features for many professional, sound applications and extensions offered by third parties.
OSC remote control enables reading and modifying (when possible) all parameters included in the `ae_data` and `ce_data JSON` structures, all part of the main parameter tree.

WING OSC server implementation complies with the OSC standard[8] and proposes several ways to access data, parameters, and features. As all OSC compliant servers, the WING OSC server runs in the console and will reply to UDP on a specific port: **2223**.
When using standard `UDP` communication, clients will be replied onto their calling port. If needed, a specific feature enables WING to reply to a `UDP port` specified by the connected client, as explained later in this document.

OSC Data Types

In compliance with the OSC standard, WING supports the following types:

- `int32` (32bits, bi-endian),
- `float32` (32bits, IEEE 754, big endian),
- `string` (non-null ASCII characters followed by a null, followed by 0-3 additional null characters to make the total number of bytes a multiple of 4),
- `blob` (An `int32` size count, followed by one or more bytes of arbitrary binary data, followed by 0-3 additional zero bytes to make the total number of bytes a multiple of 4).

As specified in the `OSC` standard, the unit of transmission of `OSC` is an `OSC Packet`. Any application that sends `OSC Packets` is an `OSC Client`; WING embeds and runs an `OSC Server`.

An `OSC Packet` consists of its contents, a contiguous block of binary data, and its size, the number of 8-bit bytes that comprise the contents. The size of an `OSC packet` is always a multiple of 4.
In the case of WING, the contents of an `OSC packet` is always an `OSC Message`, i.e. `OSC Bundles` are not supported. Note that wildcards '`?`' and '`*`' in Address Patterns are reserved for special cases.
An `OSC Message` consists of an `OSC Address Pattern` followed by an `OSC Type Tag String` followed by zero or more `OSC Arguments`. Some older implementations of `OSC` may omit the `OSC Type Tag` string and WING supports this.

- `OSC Address Patterns` always start with the character '`/`'.
- `OSC Type Tags` can be `i, f, s, b` for int32, float32, string and blob, respectively
- `OSC Arguments` consist in a single or a contiguous sequence of the binary representations of each argument

The maximum `UDP` packet size is 32k bytes.

8 See http://opensoundcontrol.org/spec-1_0

WING OSC Messages

In the following paragraphs, we assume a communication link exists between WING and a client program, and communications take place with a WING console at a known `IP` address, using `UDP` on port `2223`.

All along this document, the character '`~`' will represent a `NULL` byte (`\0`). Patterns `->W` and `W->` represent data sent to WING and data received from WING followed by the actual number of bytes transmitted or received, respectively. To generate and test the OSC patterns listed in these pages, we used `wosc`[9], a command-line tool specifically designed to operate with WING OSC.

Retrieving WING console information can be completed by sending the OSC Address Pattern "`/?`"

```
->W,     4 B: /?~~
W->,    80 B: /?~~,s~~WING,192.168.1.71,PGM,ngc-full,NO_SERIAL,1.07.2-40-g1b1b292b:develop~~~~
```

The actual bytes exchanged are displayed below (OSC is a binary protocol)

```
->W,     4 B: 2f3f0000
W->,    80 B:
2f3f00002c73000057494e472c3139322e3136382e312e37312c50474d2c6e67632d66756c6c2c4e4f5f53455249414c2c
312e30372e322d34302d673162316232323932623a646576656c6f7000000000
```

The line below is using a more compliant OSC format, and will result in the same answer

```
->W,     8 B: /?~~,~~~
```

Reading (Get) Parameter and Node data

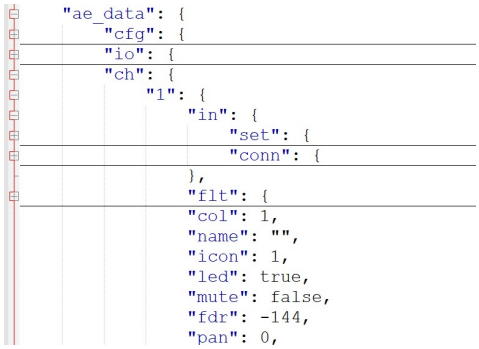
There are two main ways to gain access to WING data: using one-parameter-at-a-time or using "**nodes**".

WING "nodes" are a great way to access multiple parameters at a time, and therefore maximize communication bandwidth with the console. Nodes are represented as **string** OSC Data Type and are zero terminated (`\0` byte ending the string).
Nodes are also a good way to discover WING parameters, as they offer easy access to the full map of the `JSON` internal data structures.
We show below WING's first layer of `JSON` structure, and starting at the root, retrieved using OSC.

```
->W,     4 B: /~~~
W->,   116 B:
/~~~,sssssssssssssss~~~$stat~~~cfg~$syscfg~io~~ch~~aux~bus~main~~~~mtx~dca~mgrp~~~~fx~~cards~~~pl
ay~~~~rec~$ctl~~~~
```

Retrieving a WING single parameter is quite easy: You must ensure your OSC request points to a leaf of the `JSON` structure (i.e. there is no more hierarchy data after the current one). This is for example the case for the fader value of a channel strip, or its mute state. Channel Strip 1 fader is represented as follows:

[9] `wosc` can be found as a free tool at https://x32ram.com/downloads



Or "ch"/"1"/"fdr", which translates to OSC Address Pattern /ch/1/fdr:

```
->W,    12 B: /ch/1/fdr~~~
W->,    32 B: /ch/1/fdr~~~,sff~~~~-oo~[0.0000][-144.0000]
```

In the example above, the data [0.0000][-144.0000] are ascii representation of two 32bits big-endian float data values, each coded on 4 bytes as binary. The binary data actually received is as shown below, and in order to ease the reading of numerical information in this document, we use readable values in brackets rather than the actual binary data. The color highlights are there to help distinguish data elements.

```
W->,    32 B: 2f63682f312f666472000000 2c73666600000000 2d6f6f00 00000000 c3100000
```

Depending on the OSC Address Pattern, WING returns ',s' for strings or enums, ',sff' (ascii, raw, float value) for floats, ',sfi' (ascii, raw, int value) for ints. In the example above, fader position is a float and WING returns the ascii representation, the raw [0.0..1.0] data and the actual float value in dB.
Similarly, requesting the mute state of channel strip 1 would return:

```
->W,    12 B: /ch/1/mute~~
W->,    32 B: /ch/1/mute~~,sfi~~~~1~~~[1.0000][     1]
W->,    32 B: 2f63682f312f6d7574650000 2c7366690000000031000000 3f800000 00000001
```

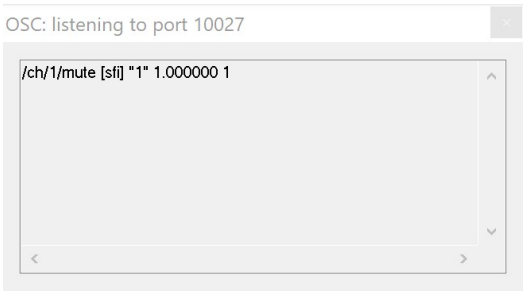
It should be noted that WING will accept both OSC path or the native hash data for representing nodes or parameters; Indeed, all nodes and parameters in the console are assigned a binary address (a hash) as explained in the chapter on native interface to the console. For example, the channel 1 mute command above can be sent as OSC Address Patterns /ch/1/mute~~ as shown, or /#f50f69f8~~, and would return the same data as shown above. 0xf50f69f8 is the hash for command "Channel 1 mute". The full set of WING hash values can be discovered by recursively traversing the JSON tree of WING nodes/commands, using the native binary interface or OSC protocol, but it is generally more convenient to use the more standard OSC node notation, rather than hexadecimal hash values to address the console features.

Receiving OSC data on a specific port

Some OSC programs will request that data is returned on a specific port rather than being sent back to the port used by the requesting client for sending data. To enable this capability, WING OSC includes an optional, special notation for all OSC commands:
Any OSC command can be prefixed with the /%<port>, with <port> in the form "12345" to enable receiving the expected answer onto the specified port number. For example, the OSC request:

```
->W,    20 B: /%10027/ch/1/mute~~~
```

Will receive the expected reply from WING on port 10027, as shown below, using a sniffer program on said port. The IP does not change.



Writing (Set) Parameter and Node data

Single Parameters

OSC can be used to set or modify WING data. Taking the fader and mute examples above, we can modify their respective values using OSC commands, sending string, big-endian int32 or big-endian float32 with the corresponding OSC Type Tag following the OSC Address Pattern respective of the parameter to change.
WING does not echo data sent over `UDP` by the client application. The client application may nevertheless be notified with an OSC event in case of an error.
Individual parameters can be strings, integer, or floats; WING OSC server implementation enables to use several data types and will manage the conversion to ensure proper value setting inside the console. For example, fader position is a floating-point internal value. It can be set as a string or a float using the following OSC commands (in this example setting channel 2 fader position to -2 or -3dB):

```
->W,    20 B: /ch/2/fdr~~~,s~~-2~~
->W,    12 B: /ch/2/fdr~~~
W->,    36 B: /ch/2/fdr~~~,sff~~~~-2.0~~~~[0.7000][-2.0000]

->W,    20 B: /ch/2/fdr~~~,f~~[-3.0000]
->W,    12 B: /ch/2/fdr~~~
W->,    36 B: /ch/2/fdr~~~,sff~~~~-3.0~~~~[0.6750][-3.0000]
```

Special case: Toggle

Wing OSC implements a specific option for toggling `[0..1]` OSC integer values for int Type Tags;
This can be quite useful to change a value without first having to read it and test its current value before sending back `0` or `1` accordingly. By sending a `-1` to an OSC Command with and integer OSC Type Tag that can only accept values `0` or `1`, the value of the parameter will toggle between `0` and `1`. For example:

```
->W,    20 B: /ch/1/mute~~,i~~[     -1]
```

Will mute channel 1 it is was unmuted, and unmute channel 1 if it was muted

Enumerated strings

One of the data WING uses is “enumerated strings”, or the choice of one string in a list of elements to represent a specific state or attribute value. For example, `/$ctl/user/1/1/enc/mode` can be any of the following strings: `OFF, FDR, PAN, DCA, SSND, FSND, FX, DAWMCU, MON, MIDICC, SD A`, or `SD B`

This can be set via a string OSC tag, as shown below if one wants to set the `mode` parameter to `FX`:

```
/$ctl/user/1/1/enc
->W,    20 B: /$ctl/user/1/1/enc~~
W->,    52 B: /$ctl/user/1/1/enc~~,sss~~~~mode~~~~name~~~~$fname~~
->W,    24 B: /$ctl/user/1/1/enc/mode~
W->,    32 B: /$ctl/user/1/1/enc/mode~,s~~OFF~
/$ctl/user/1/1/enc/mode ,s FX
```

```
->W,    32 B: /$ctl/user/1/1/enc/mode~,s~~FX~~
/$ctl/user/1/1/enc/mode
->W,    24 B: /$ctl/user/1/1/enc/mode~
W->,    32 B: /$ctl/user/1/1/enc/mode~,s~~FX~~
```

But it can also be set as an int OSC tag, using the index of the list corresponding to the targeted value; in the example above, `FX` sits at index `6` in the list of `10` strings; This enables us to use the following OSC command to set the encoder mode to `FX`:

```
/$ctl/user/1/1/enc
->W,    20 B: /$ctl/user/1/1/enc~~
W->,    52 B: /$ctl/user/1/1/enc~~,sss~~~~mode~~~~name~~~~$fname~~
->W,    24 B: /$ctl/user/1/1/enc/mode~
W->,    32 B: /$ctl/user/1/1/enc/mode~,s~~OFF~
/$ctl/user/1/1/enc/mode ,i 6
->W,    32 B: /$ctl/user/1/1/enc/mode~,i~~[     6]
/$ctl/user/1/1/enc/mode
->W,    24 B: /$ctl/user/1/1/enc/mode~
W->,    32 B: /$ctl/user/1/1/enc/mode~,s~~FX~~
```

One can also note the extendibility character of WING nodes; indeed, after the previous command, the user 1/1 encoder has additional parameters:

```
/$ctl/user/1/1/enc
->W,    20 B: /$ctl/user/1/1/enc~~
W->,    60 B: /$ctl/user/1/1/enc~~,sssss~~mode~~~~name~~~~$fname~~fx~~par~
```

Node Data

WING nodes can also be used to set multiple values with using a single OSC "`/`" command, and offer a simple yet effective way to navigate within the hierarchical structure of `JSON` data. Say you want/need to set fader and mute values to -1 dB, 0 dB, `OFF` and `ON` for channels 1 and 2; This can be achieved in a single OSC request using the following syntax:

```
->W,    44 B: /~~~,s~~/ch.1.fdr=-1,mute=0,.2.fdr=0,mute=1~
```

Or setting channel 1 fader and mute values to 10 dB and ON, and setting bus 1 fader to 5 dB:

```
->W,    44 B: /~~~,s~~/ch.1.fdr=10,mute=1,/bus.1.fdr=5~~~~
```

As shown above, each parameter group is separated by a '`,`' character, the '`/`' character represents the root of the `JSON` parameter tree, and '`.`' characters are used to navigate up and down within the `JSON` parameter tree. The console will reply with `/*~~,s~~OK~~` if the command was accepted, or one of the following:

```
/*~~,s~~NODE NOT FOUND~~
/*~~,s~~VALUE ERROR~~~~~
/*~~,s~~BUFFER OVERFLOW~
/*~~,s~~NODE IS NOT PAR~
/*~~,s~~INCOMPLETE DATA~
/*~~,s~~STACK EMPTY~~~~~
```

if an error occurred during the execution of the command.

Note: Nodes can return large amounts of data; as a result, some nodes cannot be returned using `OSC/UDP` as they would overflow the `32kB UDP` buffer limitation; In such situation, WING will return an error OSC message event.

Some nodes examples are provided below:

```
->W,    12 B: /ch/1/fdr~~~
```

```
W->,   32 B: /ch/1/fdr~~~,sff~~~~-oo~[0.0000][-144.0000]
->W,   12 B: /ch/1/mute~~
W->,   32 B: /ch/1/mute~~,sfi~~~~1~~~[1.0000][     1]
->W,   12 B: /ch/2/fdr~~~
W->,   32 B: /ch/2/fdr~~~,sff~~~~-oo~[0.0000][-144.0000]
->W,   12 B: /ch/2/mute~~
W->,   32 B: /ch/2/mute~~,sfi~~~~0~~~[0.0000][     0]

->W,   44 B: /~~~,s~~/ch.1.fdr=-1,mute=0,.2.fdr=0,mute=1~
W->,   12 B: /*~~,s~~OK~~

->W,   12 B: /ch/1/fdr~~~
W->,   36 B: /ch/1/fdr~~~,sff~~~~-1.0~~~~[0.7250][-1.0000]
->W,   12 B: /ch/1/mute~~
W->,   32 B: /ch/1/mute~~,sfi~~~~0~~~[0.0000][     0]
->W,   12 B: /ch/2/fdr~~~
W->,   32 B: /ch/2/fdr~~~,sff~~~~0.0~[0.7500][0.0000]
->W,   12 B: /ch/2/mute~~
W->,   32 B: /ch/2/mute~~,sfi~~~~1~~~[1.0000][     1]
```

Nodes can also be located deeper in the `JSON` structure tree. For example, changing a single parameter in the node channel 1 [”`/ch/1`”] can be done as shown below:

```
->W,   20 B: /ch/1~~~,s~~fdr=3~~~
W->,   16 B: /ch/1*~~,s~~OK~~

->W,   12 B: /ch/1/fdr~~~
W->,   32 B: /ch/1/fdr~~~,sff~~~~3.0~[0.8250][3.0000]
->W,   12 B: /ch/1/mute~~
W->,   32 B: /ch/1/mute~~,sfi~~~~0~~~[0.0000][     0]
```

The OSC command is replied to with an OK status if execution went well; error messages can be returned too, as explained earlier.

The same type of command can be used to set/change several parameters at once; For example, fader and mute values of channel 1 can be done as follows:

```
->W,   28 B: /ch/1~~~,s~~fdr=4,mute=1~~~~
W->,   16 B: /ch/1*~~,s~~OK~~

->W,   12 B: /ch/1/fdr~~~
W->,   32 B: /ch/1/fdr~~~,sff~~~~4.0~[0.8500][4.0000]
->W,   12 B: /ch/1/mute~~
W->,   32 B: /ch/1/mute~~,sfi~~~~1~~~[1.0000][     1]
```

Special Node Type/Arguments

There are three special tag/argument that are specifically implemented for nodes. They enable listing the complete set of data, parameter description, and description including values for the node provided as OSC address pattern. The arguments to use are ‘`*`’, ‘`?`’, and ‘`#`’, respectively. Examples of use are provided below, applied to OSC address pattern `/fx/1` when no effect is loaded to keep the description as short as possible.

Node data dump:

When using this format, the data returned will strictly correspond to what would be saved in a snap file; Read-only and temporary data are not returned.

```
/fx/1 ,s *
->W,   16 B: /fx/1~~~,s~~*~~~
```

```
W->,   32 B: /fx/1~~~,s~~mdl=NONE,fxmix=100,~
```

Node parameter description:

```
/fx/1 ,s ?
->W,   16 B: /fx/1~~~,s~~?~~~
W->,  696 B: /fx/1~~~,s~~  mdl            list [NONE, EXT, HALL, ROOM, CHAMBER, PLATE, CONCERT,
AMBI, V-ROOM, V-REV, V-PLATE, GATED, REVERSE, DEL/REV, SHIMMER, SPRING, DIMCRS, CHORUS, FLANGER,
ST-DL, TAP-DL, TAPE-DL, OILCAN, BBD-DL, PITCH, D-PITCH, VSS3, BPLATE, GEQ, PIA, DOUBLE, PCORR,
LIMITER, DE-S2, ENHANCE, EXCITER, P-BASS, ROTARY, PHASER, PANNER, TAPE, MOOD, SUB, RACKAMP,
UKROCK, ANGEL, JAZZC, DELUXE, BODY, SOUL, E88, E84, F110, PULSAR, MACH4, C5-CMB, SUB-M, V-IMG,
SPKMAN, DEQ3, *EVEN*, *SOUL*, *VINTAGE*, *BUS*, *MASTER*]~  fxmix          lin [0 .. 100 %], 101
steps~  $esrc          int [0 .. 400]~  $emode         list [M, ST, M/S]~  $a_chn         int [0
.. 76]~  $a_pos         int [0 .. 1]~~~~
```

Node description including values:

```
/fx/1 ,s #
->W,   16 B: /fx/1~~~,s~~#~~~
W->,  816 B: /fx/1~~~,s~~  mdl            NONE                list [NONE, EXT, HALL, ROOM,
CHAMBER, PLATE, CONCERT, AMBI, V-ROOM, V-REV, V-PLATE, GATED, REVERSE, DEL/REV, SHIMMER, SPRING,
DIMCRS, CHORUS, FLANGER, ST-DL, TAP-DL, TAPE-DL, OILCAN, BBD-DL, PITCH, D-PITCH, VSS3, BPLATE,
GEQ, PIA, DOUBLE, PCORR, LIMITER, DE-S2, ENHANCE, EXCITER, P-BASS, ROTARY, PHASER, PANNER, TAPE,
MOOD, SUB, RACKAMP, UKROCK, ANGEL, JAZZC, DELUXE, BODY, SOUL, E88, E84, F110, PULSAR, MACH4,
C5-CMB, SUB-M, V-IMG, SPKMAN, DEQ3, *EVEN*, *SOUL*, *VINTAGE*, *BUS*, *MASTER*]~  fxmix
100                 lin [0 .. 100 %], 101 steps~  $esrc          0               r/o int [0 ..
400]~  $emode         M               r/o list [M, ST, M/S]~  $a_chn         0               r/o
int [0 .. 76]~  $a_pos         0               r/o int [0 .. 1]~~~~
```

As a second example, we give below the node data dump for OSC address pattern `/ch/1`, when loaded with default values after init:

```
/ch/1 ,s *
->W,   16 B: /ch/1~~~,s~~*~~~
W->, 2156 B:
/ch/1~~~,s~~in.set.srcauto=0,altsrc=0,inv=0,trim=0.0,bal=0.0,dlymode=M,dly=0.1,dlyon=0,.conn.grp=L
CL,in=1,altgrp=OFF,altin=1,..flt.lc=0,lcf=100.2,lcs=24,hc=0,hcf=10k02,hcs=12,tf=0,mdl=TILT,tilt=0.
00,.clink=1,col=1,name=,icon=0,led=1,mute=0,fdr=-oo,pan=0,wid=100,solosafe=0,mon=A,proc=GEDI,ptap=
5,peq.on=0,1g=0.0,1f=100,1q=1.00,2g=0.0,2f=999,2q=1.00,3g=0.0,3f=10k0,3q=1.00,.gate.on=0,mdl=GATE,
thr=-40.0,range=40.0,att=10,hld=10,rel=199,acc=0,ratio='1:3',.gatesc.type=OFF,f=1k0,q=2.00,src=SEL
F,tap=IN,.eq.on=0,mdl=STD,mix=100,lg=0.0,lf=80.2,lq=1.00,leq=SHV,1g=0.0,1f=200.0,1q=1.00,2g=0.0,2f
=601.4,2q=1.00,3g=0.0,3f=1k50,3q=1.00,4g=0.0,4f=3k99,4q=1.00,hg=0.0,hf=12k00,hq=1.00,heq=SHV,.dyn.
on=0,mdl=COMP,mix=100,gain=0.0,thr=-10.0,ratio=3.0,knee=3,det=RMS,att=50,hld=20,rel=153,env=LOG,au
to=1,.dynxo.depth=6.0,type=OFF,f=1k0,.dynsc.type=OFF,f=1k0,q=2.00,src=SELF,tap=IN,.preins.on=0,ins
=NONE,.main.1.on=1,lvl=0.0,pre=0,.2.on=0,lvl=0.0,pre=0,.3.on=0,lvl=0.0,pre=0,.4.on=0,lvl=0.0,pre=0
,..send.1.on=0,lvl=-oo,pon=0,mode=PRE,plink=0,pan=0,.2.on=0,lvl=-oo,pon=0,mode=PRE,plink=0,pan=0,.
3.on=0,lvl=-oo,pon=0,mode=PRE,plink=0,pan=0,.4.on=0,lvl=-oo,pon=0,mode=PRE,plink=0,pan=0,.5.on=0,l
vl=-oo,pon=0,mode=PRE,plink=0,pan=0,.6.on=0,lvl=-oo,pon=0,mode=PRE,plink=0,pan=0,.7.on=0,lvl=-oo,p
on=0,mode=PRE,plink=0,pan=0,.8.on=0,lvl=-oo,pon=0,mode=PRE,plink=0,pan=0,.9.on=0,lvl=-oo,pon=0,mod
e=GRP,plink=1,pan=0,.10.on=0,lvl=-oo,pon=0,mode=GRP,plink=1,pan=0,.11.on=0,lvl=-oo,pon=0,mode=POST
,plink=1,pan=0,.12.on=0,lvl=-oo,pon=0,mode=POST,plink=1,pan=0,.13.on=0,lvl=-oo,pon=0,mode=POST,pli
nk=1,pan=0,.14.on=0,lvl=-oo,pon=0,mode=POST,plink=1,pan=0,.15.on=0,lvl=-oo,pon=0,mode=POST,plink=1
,pan=0,.16.on=0,lvl=-oo,pon=0,mode=POST,plink=1,pan=0,.MX1.on=0,lvl=-oo,pon=0,mode=PRE,plink=0,pan
=0,.MX2.on=0,lvl=-oo,pon=0,mode=PRE,plink=0,pan=0,.MX3.on=0,lvl=-oo,pon=0,mode=PRE,plink=0,pan=0,.
MX4.on=0,lvl=-oo,pon=0,mode=PRE,plink=0,pan=0,.MX5.on=0,lvl=-oo,pon=0,mode=PRE,plink=0,pan=0,.MX6.
on=0,lvl=-oo,pon=0,mode=PRE,plink=0,pan=0,.MX7.on=0,lvl=-oo,pon=0,mode=PRE,plink=0,pan=0,.MX8.on=0
,lvl=-oo,pon=0,mode=PRE,plink=0,pan=0,..tapwid=100,postins.on=0,mode=FX,ins=NONE,w=0.0,.tags=,~~~~
```

OSC: Special Cases

JSON Structure dynamic changes

As parameters get changed on the WING console, its `JSON` structure tree evolves to reflect the changes; This can be a specific parameter that when changing to an `ON` state, offers new capabilities in the audio chain, or in the way the console will react.
It is also typical of **effects** and **plugins**: WING consoles support dynamic allocation of effect or plugins that can generate large changes within the default `JSON` tree. As already mentioned, WING nodes are a great way to list the parameters available for a given effect and therefore a way to get and possibly set effect parameter values.
The WING effects and plugins, and their respective parameters are listed later in this document[10].

The OSC commands below show how you can access effects slots, allocate an effect, and list parameters and later modify effect parameter values.

Accessing effects with currently no effect loaded in effect slot 1, listing the effect node:

```
->W,     4 B: /fx~
W->,    88 B:
/fx~,sssssssssssssss~~~1~~~2~~~3~~~4~~~5~~~6~~~7~~~8~~~9~~~10~~11~~12~~13~~14~~15~~16~~

->W,     8 B: /fx/1~~~
W->,    60 B: /fx/1~~~,ssssss~mdl~fxmix~~~$esrc~~~$emode~~$a_chn~~$a_pos~~
->W,    12 B: /fx/1/mdl~~~
W->,    24 B: /fx/1/mdl~~~,s~~NONE~~~~
```

Loading a PIA effect in effect slot 1:

```
->W,    20 B: /fx/1/mdl~~~,s~~pia~
->W,    12 B: /fx/1/mdl~~~
W->,    20 B: /fx/1/mdl~~~,s~~PIA~
```

PIA effect is now loaded, listing the effect Node gives a different set of parameters:

```
->W,     8 B: /fx/1~~~
W->,   120 B:
/fx/1~~~,sssssssssssssssss~mdl~fxmix~~~$esrc~~~$emode~~$a_chn~~$a_pos~~mix~g~~~31~~63~~125~250~50
0~1k~~2k~~4k~~8k~~16k~
```

We can now get/set effect 1 PIA parameters, for example the 125Hz band:

```
->W,    12 B: /fx/1/125~~~
W->,    32 B: /fx/1/125~~~,sff~~~~0.0~[0.5000][0.0000]
```

The 125Hz band is at 0dB, change it to 10dB and verify the change:

```
->W,    20 B: /fx/1/125~~~,f~~[10.000]
->W,    12 B: /fx/1/125~~~
W->,    36 B: /fx/1/125~~~,sff~~~~10.0~~~~[0.9233][10.000]
```

[10] Please refer to the "Effects" paragraph

OSC Tag Type 'blob' or 'binary' use

WING OSC server implementation supports the 'blob'/'binary' OSC Tag type, enabling the use of 'native' commands[11] within OSC, making it is possible with the proper information at hand to send and receive binary data.

An alternative to standard node requests (such as the request on root below) is to use binary.

```
->W,     4 B: /~~~
W->,   116 B:
/~~~,sssssssssssssss~~~$stat~~~cfg~$syscfg~io~~ch~~aux~bus~main~~~~mtx~dca~mgrp~~~~fx~~cards~~~play~~~~rec~$ctl~~~~
```

Binary types typically apply on WING nodes to retrieve the internal binary equivalent of the `JSON` tree level respective of a WING node.
Shown below is a request at root level using the native commands part of the binary data [all bytes sent shown as hex data]

```
/ ,b dd
```

Data actually sent (in hex): `->W,    16 B: 2f0000002c62000000000001dd000000`
WING's reply is:

```
W->,   440 B: /~~~,b~~425 bytes:
df00180000000097a00439000005247374617405535441544500 00df001100000000edca7af900000363666700000 0df00
1500000000f89818a60000072473797363666700000 0df001300000000294f7794000002696f03492f4f0000df00170000
0000070b10139000002636807434 8414e4e454c0000df001c000000008fa3078d0000036175780b415558204348414e4e45
4c0000df001400000000f46c185e000003627573034255530000df00160000000004d3a3a80000046d61696e044d41494e
0000df001700000000f82a5af20000036d7478064d41545249580000df001400000000e313aeff000003646361034434 1
0000df001c00000000d252398b0000046d6772700a4d5554452047524f55500000df001700000000473c91340000026678
074546464543545300 00df002200000000b4296fc900000563617264730f455850414e53494f4e2043415244530000df00
180000000057297a28000004706c617906504c415945520000df001900000000fab1762c000003726563085245434f5244
45520000df001900000000cbb9514300000424637 46c07434f4e54524f4c0000de
```

Lots of information are returned either as string, or more often as blob/binary. In the reply above, after each '`df`' byte is a data length on two bytes, immediately followed by the binary address (the hash) where a node, parameter, or subtree data can be found. For example, the subtree entry for channel (`/ch`) can be found at address/hash `70b10139`

An example on retrieving the DAW node (hash is `df17c242`, part of the `$ctl` subtree) is shown below. Sending the OSC blob:

```
/$ctl/daw ,b dd
```
or
```
/ ,b d7df17c242dd
```

Respectively translate in the following binary data being sent to the console:

```
->W,   24 B: 2f2463746c2f6461770000002c62000000000001dd000000
```
or
```
->W,   20 B: 2f0000002c62000000000006d7df17c242dd0000
```

To which the console replies with (it can also reply with one of the errors listed earlier in the OSC chapters):

```
W->,   876 B: /$ctl/daw~~~,b~~856 bytes:
df0022df17c2423cb129d50000026f6e0a44415720454e41424c450040000000000000001df0028df17c2424e5c7f3400
0004636f6e6e0a434f4e4e454354494f4e005000020344494e000355534200df0027df17c242e5681680000004656d756c
09454d554c4154494f4e00500002034d43550003485549 00df0071df17c24242701ca9000006636f6e6669670000500004
02434314435553544f4d20434f4e54524f4c53204f4e4c59044d5354520a53494e474c45204d4355084d53545231455854
```

[11] Detail information on native commands is provided in a separate chapter

```
0e4d4355202b20455854454e444552084d53545232455854114d4355202b20327820455854454e444552df002edf17c242
ae1538a40000046363757014555345205550504552204343204 64f52204441570040000000000000000001df0035df17c242
9fa4e7320000066469736a6f671944495341424c4520574845454c20445552494e4720504c41590040000000000000000001
df0097df17c242892e512d0000067072657365741 24c415354204c4f414445442050524553455400500008012d012d0663
756261736504355424154534 6c697665044c495645066c6f67696378074c4f4749432058066e75656e646f064e55454e
444f0870726f746f6f6c730950524f20544f4f4c530672656170657206524541504552097374756469 6f6f6e650a535455
44494f204f4e45df001fdf17c242beefaeab000003246f6e06444157204f4e024000000000000000001df0027df17c24296
31559f000006246270616765 0b425554544f4e2050414745004000000000000000004df0031df17c242012dc54600000924
62746e746f7563681242544e53454c20464144455220544f554348004000000000000000001df002adf17c242775c19c200
00082462746e76706f740c42544e53454c20562d504f5400400000000000000000001df002ddf17c24242aeb92800000a2462
746e7265637264790d42544e53454c2052454352445900400000000000000000001df0029df17c242fccfbe070000082462 74
6e6175746f0b42544e53454c204155544f004000000000000000001df002adf17c24285cdce3f0000082462746e7673656c
0c42544e53454c20562d53454c004000000000000000001df002ddf17c24215abd96800000a2462746e696e736572740d42
544e53454c20494e534552540040000000000000001de
```

The above is more difficult to read than the more standard way of retrieving the node, but contains more information:

```
->W,  12 B: /$ctl/daw~~~
W->, 156 B:
/$ctl/daw~~~,sssssssssssss~on~~conn~~~~emul~~~~config~~ccup~~~~preset~~$on~$bpage~~$btntouch~~~$b
tnvpot~~~~$btnrecrdy~~$btnauto~~~~$btnvsel~~~~$btninsert~~
```

Matching the two representations tell us that:
`daw/on` is at binary address `3cb129d5`,
`daw/conn` at `4e5c7f34`,
`daw/emul` at `e5681680`,
`daw/config` at `42701ca9`,
`daw/ccup` at `ae1538a4`,
`daw/preset` at `892e512d`,
`daw/$on` at `beefaeab`,
and so on (highlighted values above).

The blob /binary Type Tag can also be used to execute native/binary commands. Using for example the `daw/$on` hash/binary address value of `beefaeab,` we can set the console in and out of DAW mode, as if one would have pressed the DAW button.
For example, sending any of the following commands will set DAW mode ON:

```
/ ,b d7beefaeab01
->W,   20 B: /~~~,b~~6 bytes: d7beefaeab01
W->,   12 B: /*~~,s~~OK~~
/$ctl/daw/$on ,b 01
->W,   28 B: /$ctl/daw/$on~~~,b~~1 bytes: 01~~~
W->,   12 B: /*~~,s~~OK~~
```

In the binary data sent with the line above, the segment `01` is equivalent to asking the value of the parameter to be set using a 32bit integer with value 1.

The following lines are requesting to turn OFF DAW mode:

```
/ ,b d7beefaeab00
->W,   20 B: /~~~,b~~6 bytes: d7beefaeab00
W->,   12 B: /*~~,s~~OK~~
/$ctl/daw/$on ,b 00
->W,   28 B: /$ctl/daw/$on~~~,b~~1 bytes: 00~~~
W->,   12 B: /*~~,s~~OK~~
```

In both blob Type Tag commands above, the console replies with a blob. Depending on the cases, it can also return strings.

As seen above, the Tag Type blob can be used to retrieve the description of WING parameters when using the native command 'data description' a.k.a. **'dd'**; In an example below, still using the DAW ON state, we can get the data using the following command:

```
/$ctl/daw/$on ,b dd
->W,   28 B: /$ctl/daw/$on~~~,b~~1 bytes: dd~~~
```

WING returns the following which includes the hash value for `/$ctl/daw/$on` and its full description:

```
W->,   60 B: /$ctl/daw/$on~~~,b~~35 bytes:
df001fdf17c242beefaeab0000003246f6e06444157204f4e0040000000000000001de
parse 35 bytes node
    len: 31, parent: df17c242, hash: beefaeab, index: 0, flags: 0040
    name: $on longname: DAW ON, type: <int> [0..1]

End node
```

The blob Tag Type can be used to retrieve the value of WING parameters when using the native command 'data request', a.k.a. **'dc'**; In an example below, still using the DAW ON state, we can get the data using the following command:

```
->W,   20 B: /~~~,b~~6 bytes: d7beefaeabdc
W->,   20 B: /~~~,b~~7 bytes: d7beefaeab01de

With 01 indicating the DAW [Remote control] button is in an ON state.
```

Detailed information on the native/binary interface to WING and data value coding is provided later in this document.

Subscribing to OSC Data

There are three main types of subscription for receiving binary or OSC messages.
A single OSC subscription is active at any time, provided to the last requestor. Subscriptions must be renewed every `10 seconds` to keep the subscription alive by sending one of the 3 messages shown below.

`/*b~` (or `/*b~,~~~`) will enable receiving event driven binary messages
Binary messages are formatted exactly as the binary/native interface and therefore can be sent back to the console with no change.

Example using mutes and faders

```
->W,     4 B: /*b~
W->,    32 B: /~~~,b~~20 bytes: d738ae75c2d5c3100000d77e463474d5c3100000
W->,    24 B: /~~~,b~~12 bytes: d7f50f69f801d726855cd301
```

`/*s~` (or `/*s~,~~~`) will enable receiving event OSC messages
OSC messages are received as triplets of data, as previously presented[12], and shown below; Sending back data to WING will require to select one of the (up to) 3 parameters received, depending on the chosen format. The 'string' argument will always work for all messages.

Example using mutes and faders

```
->W,     4 B: /*s~
W->,    32 B: /ch/1/fdr~~~,sff~~~~-oo~[0.0000][-144.0000]
W->,    32 B: /ch/1/$fdr~~,sff~~~~-oo~[0.0000][-144.0000]
W->,    32 B: /ch/1/mute~~,sfi~~~~1~~~[1.0000][     1]
W->,    32 B: /ch/1/$mute~,sfi~~~~1~~~[0.5000][     1]
```

`/*S~` (or `/*S~,~~~`) will enable receiving event OSC messages
OSC messages are received as single tag data, as shown below; WING reports the native format of the OSC pattern (ex: '`f`' for floats, '`i`' for integers, etc.). Data received with events resulting of a `/*S~` subscription can be sent back to the console with no change.

Example using mutes and faders

```
->W,     4 B: /*S~
W->,    20 B: /ch/1/fdr~~~,f~~[-144.0000]
W->,    20 B: /ch/1/$fdr~~,f~~[-144.0000]
W->,    20 B: /ch/1/mute~~,i~~[     1]
W->,    20 B: /ch/1/$mute~,i~~[     1]
```

Using the simple forms of subscription requests will provide data from the console to the requesting IP/port. It is possible to redirect the data received from WING by prefixing the commands with a port specifier element as shown below:

`/%23456/*b~` will subscribe to binary messages, being sent by WING to port `23456`.
`/%23456/*s~` will subscribe to OSC messages, being sent by WING to port `23456`.
`/%23456/*S~` will subscribe to OSC messages, being sent by WING to port `23456`.

[12] Refer to "Writing (Set) Parameter and Node data", paragraph "Single Parameters"

WING ae_data OSC commands list

The next chapters provide an abridged[13] list of all OSC commands available for WING.

All commands and parameters below are part of the ***ae_data*** section in `JSON` snapshot files. Other console control commands part of the ***ce_data*** section in `JSON` snapshot files are described later in this document.

Status

Command	Type	Range	Text	Description
/$stat	N			Status node
/$stat/A	N			AES50 A node
/$stat/A/stat	S		-, OK, ERR, UPD	AES50 A state [RO]
/$stat/A/dev	S		128 chars max	AES50 A Device [RO]
/$stat/A/errorsc	I		0..9999	Corrected error count [RO]
/$stat/A/errorsu	I		0..9999	Uncorrected error count [RO]
/$stat/A/clrerr	I		0..1	Reset error counters
/$stat/B	N			AES50 B node
/$stat/B/stat	S		-, OK, ERR, UPD	AES50 B state [RO]
/$stat/B/dev	S		128 chars max	AES50 B Device [RO]
/$stat/B/errorsc	I		0..9999	Corrected error count [RO]
/$stat/B/errorsu	I		0..9999	Uncorrected error count [RO]
/$stat/B/clrerr	I		0..1	Reset error counters
/$stat/C	N			AES50 C node
/$stat/C/stat	S		-, OK, ERR, UPD	AES50 C state [RO]
/$stat/C/dev	S		128 chars max	AES50 C Device [RO]
/$stat/C/errorsc	I		0..9999	Corrected error count RO]
/$stat/C/errorsu	I		0..9999	Uncorrected error count [RO]
/$stat/C/clrerr	I		0..1	Reset error counters
/$stat/lock	I	0..1		Clock lock [RO]
/$stat/ppm	I	-200..200		Clock ppm [RO]
/$stat/solo	I	0..1		Solo [RO]
/$stat/sip	I	0..1		Solo In Place [RO]
/$stat/rtcerr	I	0..1		Real Time Clock Error [RO]
/$stat/time	S		12 chars max	Clock time (depending on time format) [RO]
/$stat/date	S		12 chars max	Clock date (depending on date format) [RO]
/$stat/usbstate	S		-, ERR, IDLE, BUSY	USB Player state [RO]
/$stat/usbvolname	S		20 chars max	USB Player volume name [RO]
/$stat/sc_stat	S		OK, ERR	StageConnect status [RO]
/$stat/sc_devices	S		128 chars max	StageConnect devices [RO]
/$stat/sc_upcnt	I	0..32		StageConnect upstreams [RO]
/$stat/sc_dncnt	I	0..32		StageConnect downstreams [RO]

[13] It includes the set of commands for the first element of a series. For example, `/ch/1` set of OSC commands are listed, but not `/ch/2` to `/ch/40`.

/$stat/sc_uprout	S		32 char max	StageConnect upstream routing [RO]
/$stat/rmt_a	S		16 chars max	Name of the console connected on AES50 port A [RO]
/$stat/rmt_b	S		16 chars max	Name of the console connected on AES50 port A [RO]
/$stat/rmt_c	S		16 chars max	Name of the console connected on AES50 port A [RO]

General Configuration

Command	Type	Range	Text	Description
/cfg	N			General Configuration node
/cfg/mainlink	S		OFF, 2, 2-3, 2-4	Main Link
/cfg/dcamgrp	I	0..1		DCA mutegroups (DCA mute mutes all channels assigned to DCA)
/cfg/mon	N			Monitor buses config node
/cfg/mon/1	N	1..2		Monitor bus 1 node
/cfg/mon/1/$lvl	F	-144..10	-oo..10 in 1024 steps	Monitor bus 1 level (dB)[14]
/cfg/mon/1/inv	I	0..1		Monitor bus 1 invert (polarity)
/cfg/mon/1/pan	F	-100..100	201 steps	Monitor bus 1 pan
/cfg/mon/1/wid	F	-150..150	61 steps	Monitor bus 1 width (%)
/cfg/mon/1/eq	N			Monitor bus 1 EQ node
/cfg/mon/1/eq/on	I	0..1		Monitor bus 1 EQ off/on
/cfg/mon/1/eq/lsg	F	-15..15	301 steps	Monitor bus 1 EQ low shelf gain (dB)
/cfg/mon/1/eq/lsf	F	20..2000	641 steps	Monitor bus 1 EQ low shelf frequency (Hz)
/cfg/mon/1/eq/1g	F	-15..15	301 steps	Monitor bus 1 EQ band 1 gain (dB)
/cfg/mon/1/eq/1f	F	20..20000	961 steps	Monitor bus 1 EQ band 1 frequency (Hz)
/cfg/mon/1/eq/1q	F	0.44..10	181 steps	Monitor bus 1 EQ band 1 Q
/cfg/mon/1/eq/2g	F	-15..15	301 steps	Monitor bus 1 EQ band 2 gain (dB)
/cfg/mon/1/eq/2f	F	20..20000	961 steps	Monitor bus 1 EQ band 2 frequency (Hz)
/cfg/mon/1/eq/2q	F	0.44..10	181 steps	Monitor bus 1 EQ band 2 Q
/cfg/mon/1/eq/3g	F	-15..15	301 steps	Monitor bus 1 EQ band 3 gain (dB)
/cfg/mon/1/eq/3f	F	20..20000	961 steps	Monitor bus 1 EQ band 3 frequency (Hz)
/cfg/mon/1/eq/3q	F	0.44..10	181 steps	Monitor bus 1 EQ band 3 Q
/cfg/mon/1/eq/4g	F	-15..15	301 steps	Monitor bus 1 EQ band 4 gain (dB)
/cfg/mon/1/eq/4f	F	20..20000	961 steps	Monitor bus 1 EQ band 4 frequency (Hz)
/cfg/mon/1/eq/4q	F	0.44..10	181 steps	Monitor bus 1 EQ band 4 Q
/cfg/mon/1/eq/5g	F	-15..15	301 steps	Monitor bus 1 EQ band 5 gain (dB)
/cfg/mon/1/eq/5f	F	20..20000	961 steps	Monitor bus 1 EQ band 5 frequency (Hz)
/cfg/mon/1/eq/5q	F	0.44..10	181 steps	Monitor bus 1 EQ band 5 Q
/cfg/mon/1/eq/6g	F	-15..15	301 steps	Monitor bus 1 EQ band 6 gain (dB)
/cfg/mon/1/eq/6f	F	20..20000	961 steps	Monitor bus 1 EQ band 6 frequency (Hz)
/cfg/mon/1/eq/6q	F	0.44..10	181 steps	Monitor bus 1 EQ band 6 Q
/cfg/mon/1/eq/hsg	F	-15..15	301 steps	Monitor bus 1 EQ high shelf gain (dB)
/cfg/mon/1/eq/hsf	F	50..20000	833 steps	Monitor bus 1 EQ high shelf frequency (Hz)
/cfg/mon/1/lim	F	-40..0	41 steps	Monitor bus 1 limiter level(dB)

[14] This command is considered RO on the full-size WING, and can be set for other devices where the actual surface control potentiometer is not present.

/cfg/mon/1/dly	N			Monitor bus 1 delay node
/cfg/mon/1/dly/on	I	0..1		Monitor bus 1 delay off/on
/cfg/mon/1/dly/m	F	0.1..100	1000 steps	Monitor bus 1 delay (meters)
/cfg/mon/1/dim	F	40..0	41 steps	Monitor bus 1 delay dim level (dB)
/cfg/mon/1/pfldim	F	40..0	41 steps	Monitor bus 1 PFL Dim (dB)
/cfg/mon/1/eqbdtrim	F	0..24	25 steps	Monitor bus 1 band solo trim {dB)
/cfg/mon/1/srclvl	F	-144..10	-oo..10 in 1024 steps	Monitor bus 1 source level
/cfg/mon/1/srcmix	F	-144..10	-oo..10 in 1024 steps	Monitor bus 1 source mix (dB)
/cfg/mon/1/src	S		OFF, MAIN.1..MAIN.4, MTX.1..MTX.8, BUS.1..BUS.16, AUX.1..AUX.8	Monitor bus 1 source
/cfg/mon/1/dirin	S		OFF, CH.1.. CH.40, AUX.1.. AUX.8, BUS.1..BUS.16, MAIN.1..MAIN.4, MTX.1.. MTX.8	Monitor bus 1 direct in
/cfg/mon/1/$lvlact	F	-144..10	-oo..10 in 1024 steps	Monitor bus 1 fader level [RO]
/cfg/mon/1/tags	S		Up to 80 chars	Monitor bus 1 tags
/cfg/solo	N			Solo config node
/cfg/solo/mode	S		LIVE, STUDIO, SIP	Solo mode
/cfg/solo/mon	S		PH, SPK, PH+SPK	Solo monitor
/cfg/solo/mute	I	0..1		Solo mute
/cfg/solo/$dim	I	0..1		Solo dim off/on
/cfg/solo/$mono	I	0..1		Solo mono off/on
/cfg/solo/$flip	I	0..1		Solo left and right channels flipped
/cfg/solo/chtap	S		PFL, AFL	Solo channel tap
/cfg/solo/bustap	S		PFL, AFL	Solo bus tap
/cfg/solo/maintap	S		PFL, AFL	Solo main tap
/cfg/solo/mtxtap	S		PFL, AFL	Solo matrix tap
/cfg/solo/srcsolo	S		OFF, CH39, AUX7	Source Solo Enable
/cfg/solo/$srcsolo	I	0..1		Source Solo
/cfg/solo/$srcsgrp	I	1..13		Source Solo Group
/cfg/solo/$srcsin	I	1..64		Source Solo In
/cfg/rta[15] [16]	N			RTA config node (dsp)
/cfg/rta/$src	I	1..76		RTA source [RO]
/cfg/rta/$tap	S		IN, POST, FILT, PREEQ, POSTEQ, PREFDR, GATEK, DYNK, DYNXO, PRETAP, SOLO, MON.PH, MON.SPK, FXIN, FXOUT	RTA source tap [RO]
/cfg/rta/$dec	S		SLOW, MED, FAST	RTA Decay [RO]
/cfg/rta/$det	S		PEAK, RMS	RTA Detector [RO]
/cfg/rta/rtasrc	I	0..76		*RTA source (indexed)
/cfg/rta/rtatap	S		IN, POST, FILT, PREEQ, POSTEQ, PREFDR, GATEK, DYNK, DYNXO, PRETAP, SOLO, MON.PH, MON.SPK, FXIN, FXOUT	*RTA source tap
/cfg/rta/rtadecay	S		SLOW, MED, FAST	*RTA decay
/cfg/rta/rtadet	S		PEAK, RMS, AVG	*RTA detector
/cfg/rta/rtarange	F	30, 60		*RTA range (dB)

[15] Tags (marqued with *) are only used with metering RTA and future RTA screen, as opposed to EQ (on-screen) RTA

[16] See also /$ctl/cfg/rta commands

/cfg/rta/rtagain	F	-5..50	56 steps	*RTA gain (dB)
/cfg/rta/rtaauto	I	0..1		*RTA autogain
/cfg/rta/eqdecay	S		SLOW, MED, FAST	RTA eq decay
/cfg/rta/eqdet	S		PEAK, RMS, AVG	RTA eq detector
/cfg/rta/eqrange	F	30, 60		RTA eq range (dB)
/cfg/rta/eqgain	F	-5..50	56 steps	RTA eq gain (dB)
/cfg/rta/eqauto	I	0..1		RTA eq autogain
/cfg/mtr	N			Meter config node
/cfg/mtr/$scopesrc	I	1..76		Meter scope source [RO]
/cfg/mtr/$scopetap	S		IN, POST, FILT, PREEQ, POSTEQ, PREFDR, GATEK, DYNK, DYNXO, PRETAP, SOLO, MON.PH, MON.SPK, FXIN, FXOUT	Meter scope source tap point [RO]
/cfg/mtr/scopesrc	I	0..76		Meter scope source
/cfg/mtr/scopetap	S		IN, POST, FILT, PREEQ, POSTEQ, PREFDR, GATEK, DYNK, DYNXO, PRETAP, SOLO, MON.PH, MON.SPK	Meter scope source tap point
/cfg/mtr/mtrsfc	N			Meters fader node [Setup→Surface]
/cfg/mtr/mtrsfc/in	S		PRE, POST	Meters fader section channel tap
/cfg/mtr/mtrsfc/bus	S		PRE, POST	Meters fader section bus tap
/cfg/mtr/mtrsfc/main	S		PRE, POST	Meters fader section main tap
/cfg/mtr/mtrsfc/mtx	S		PRE, POST	Meters fader section matrix tap
/cfg/mtr/mtrsfc/dca	S		PRE, POST	Meters fader section DCA tap
/cfg/mtr/mtrpage	N			Meters page node (Meters screen)
/cfg/mtr/mtrpage/in	S		PRE, POST	Meters page channels tap
/cfg/mtr/mtrpage/bus	S		PRE, POST	Meters page bus tap
/cfg/mtr/mtrpage/main	S		PRE, POST	Meters page mains tap
/cfg/mtr/mtrpage/mtx	S		PRE, POST	Meters page matrix tap
/cfg/mtr/mtrpage/dca	S		PRE, POST	Meters page DCA tap
/cfg/mtr/mainmtr	S		[MAIN.1...4, MTX.1…8, MON.PH, MON.SPK, SEL_CH	Main meter
/cfg/mtr/mainpos	S		AUTO, PRE, POST	Main position
/cfg/talk	N			Talkback config node
/cfg/talk/assign	S		OFF, CH40, AUX8	Talkback assignments
/cfg/talk/$lvl	F	-144..10	-oo..10 in 1024 steps	Talkback level (dB) [RO]
/cfg/talk/A	N			Talkback A node
/cfg/talk/A/$on	I	0..1		Talkback A off/on
/cfg/talk/A/mode	S		AUTO, PUSH, LATCH	Talkback A mode
/cfg/talk/A/mondim	I	40..0	41 steps	Talkback A monitor dim
/cfg/talk/A/busdim	F	40..0	41 steps	Talkback A bus dim
/cfg/talk/A/indiv	I	0..1		Use individual Bus/Main TB send levels
/cfg/talk/A/B1	I	0..1		Talkback A bus 1 assign
/cfg/talk/A/B2	I	0..1		Talkback A bus 2 assign
/cfg/talk/A/B3	I	0..1		Talkback A bus 3 assign

/cfg/talk/A/B4	I	0..1		Talkback A bus 4 assign
/cfg/talk/A/B5	I	0..1		Talkback A bus 5 assign
/cfg/talk/A/B6	I	0..1		Talkback A bus 6 assign
/cfg/talk/A/B7	I	0..1		Talkback A bus 7 assign
/cfg/talk/A/B8	I	0..1		Talkback A bus 8 assign
/cfg/talk/A/B9	I	0..1		Talkback A bus 9 assign
/cfg/talk/A/B10	I	0..1		Talkback A bus 10 assign
/cfg/talk/A/B11	I	0..1		Talkback A bus 11 assign
/cfg/talk/A/B12	I	0..1		Talkback A bus 12 assign
/cfg/talk/A/B13	I	0..1		Talkback A bus 13 assign
/cfg/talk/A/B14	I	0..1		Talkback A bus 14 assign
/cfg/talk/A/B15	I	0..1		Talkback A bus 15 assign
/cfg/talk/A/B16	I	0..1		Talkback A bus 16 assign
/cfg/talk/A/MX1	I	0..1		Talkback A matrix 1 assign
/cfg/talk/A/MX2	I	0..1		Talkback A matrix 2 assign
/cfg/talk/A/MX3	I	0..1		Talkback A matrix 3 assign
/cfg/talk/A/MX4	I	0..1		Talkback A matrix 4 assign
/cfg/talk/A/MX5	I	0..1		Talkback A matrix 5 assign
/cfg/talk/A/MX6	I	0..1		Talkback A matrix 6 assign
/cfg/talk/A/MX7	I	0..1		Talkback A matrix 7 assign
/cfg/talk/A/MX8	I	0..1		Talkback A matrix 8 assign
/cfg/talk/A/M1	I	0..1		Talkback A main 1 assign
/cfg/talk/A/M2	I	0..1		Talkback A main 2 assign
/cfg/talk/A/M3	I	0..1		Talkback A main 3 assign
/cfg/talk/A/M4	I	0..1		Talkback A main 4 assign
/cfg/talk/B	N			Talkback B node
/cfg/talk/B/$on	I	0..1		Talkback B off/on
/cfg/talk/B/mode	S		AUTO, PUSH, LATCH	Talkback B mode
/cfg/talk/B/mondim	F	40..0	41 steps	Talkback B monitor dim
/cfg/talk/B/busdim	F	40..0	41 steps	Talkback B bus dim
/cfg/talk/B/indiv	I	0..1		Use individual Bus/Main TB send levels
/cfg/talk/B/B1	I	0..1		Talkback B bus 1 assign
/cfg/talk/B/B2	I	0..1		Talkback B bus 2 assign
/cfg/talk/B/B3	I	0..1		Talkback B bus 3 assign
/cfg/talk/B/B4	I	0..1		Talkback B bus 4 assign
/cfg/talk/B/B5	I	0..1		Talkback B bus 5 assign
/cfg/talk/B/B6	I	0..1		Talkback B bus 6 assign
/cfg/talk/B/B7	I	0..1		Talkback B bus 7 assign
/cfg/talk/B/B8	I	0..1		Talkback B bus 8 assign
/cfg/talk/B/B9	I	0..1		Talkback B bus 9 assign
/cfg/talk/B/B10	I	0..1		Talkback B bus 10 assign
/cfg/talk/B/B11	I	0..1		Talkback B bus 11 assign
/cfg/talk/B/B12	I	0..1		Talkback B bus 12 assign
/cfg/talk/B/B13	I	0..1		Talkback B bus 13 assign
/cfg/talk/B/B14	I	0..1		Talkback B bus 14 assign
/cfg/talk/B/B15	I	0..1		Talkback B bus 15 assign
/cfg/talk/B/B16	I	0..1		Talkback B bus 16 assign
/cfg/talk/B/MX1	I	0..1		Talkback B matrix 1 assign
/cfg/talk/B/MX2	I	0..1		Talkback B matrix 2 assign
/cfg/talk/B/MX3	I	0..1		Talkback B matrix 3 assign

/cfg/talk/B/MX4	I	0..1		Talkback B matrix 4 assign
/cfg/talk/B/MX5	I	0..1		Talkback B matrix 5 assign
/cfg/talk/B/MX6	I	0..1		Talkback B matrix 6 assign
/cfg/talk/B/MX7	I	0..1		Talkback B matrix 7 assign
/cfg/talk/B/MX8	I	0..1		Talkback B matrix 8 assign
/cfg/talk/B/M1	I	0..1		Talkback B main 1 assign
/cfg/talk/B/M2	I	0..1		Talkback B main 2 assign
/cfg/talk/B/M3	I	0..1		Talkback B main 3 assign
/cfg/talk/B/M4	I	0..1		Talkback B main 4 assign
/cfg/amix	N			Automixing node
/cfg/amix/x	I	0..1		Automix X group enable
/cfg/amix/y	I	0..1		Automix Y group enable

System Settings

Command	Type	Range	Text	Description
/$syscfg	N			System configuration node
/$syscfg/consolename	S		16 chars max	Console name
/$syscfg/logflags	S		256 char max	Log flags
/$syscfg/ipmode	S		DHCP, STATIC	IP Mode
/$syscfg/ip0	I	0..255		IP first number
/$syscfg/ip1	I	0..255		IP second number
/$syscfg/ip2	I	0..255		IP third number
/$syscfg/ip3	I	0..255		IP fourth number
/$syscfg/msk0	I	0..255		IP mask first number
/$syscfg/msk1	I	0..255		IP mask second number
/$syscfg/msk2	I	0..255		IP mask third number
/$syscfg/msk3	I	0..255		IP mask fourth numbe
/$syscfg/gw0	I	0..255		IP gateway first number
/$syscfg/gw1	I	0..255		IP gateway second number
/$syscfg/gw2	I	0..255		IP gateway third number
/$syscfg/gw3	I	0..255		IP gateway fourth number
/$syscfg/$ipapply	I	0..1		IP applied
/$syscfg/$firmware	S		64 chars max	Firmware version number [RO]
/$syscfg/$serial	S		32 chars max	Serial number [RO]
/$syscfg/$cnscfg	S		64 chars max	Console configuration/build type string [RO], typically start with “wing”, “wing-rack”, “wing-compact”
/$syscfg/$cnsmdl	S		32 chars max	Console Model right to Console Name in Setup screen [RO], typically “ngc-full” for the full sized desk, can also be “wing-bk”, ”wing-rack”, “wing-compact”
/$syscfg/$chwversion	S		32 chars max	Main board HW version [RO]
/$syscfg /tcplock	I	0..1		Prevent modifications from TCP input
/$syscfg/usbh_spd	S		FS, HS	USB driver speed setting Full Speed, High Speed[17]
/$syscfg/$usbspd_act	S		FS, HS	USB driver speed setting Full Speed, High Speed [RO]
/$syscfg/eth/cfg	S		SEPARATE, SWITCHED	Optional module Ethernet mode
/$syscfg/opt_mod	S		NONE, DANTE, WSG, ACOM	Installed optional module [RO]

[17] When in FS, record is limited to 2 tracks/16bits. 4 tracks/24bits playing at once from USB stick may be affected; USB 3.1 capable memory sticks are recommended.

Input/Output Settings

Command	Type	Range	Text	Description
/io	N			Input/Output node
/io/altsw	I	0..1		Main/Alt switch
/io/autoaltovr	I	0..1		Global Input Select Override
/io/in	N			Input node
/io/in/LCL	N			Local Input node
/io/in/LCL/1	N	1..24[18]		Local Input 1 node
/io/in/LCL/1/mode	S		M, ST, M/S	Local Input 1 mode
/io/in/LCL/1/g	F	-3..45.5	98 steps	Local Input 1 gain (dB)
/io/in/LCL/1/vph	I	0..1		Local Input 1 phantom
/io/in/LCL/1/mute	I	0..1		Local Input 1 mute
/io/in/LCL/1/pol	I	0..1		Local Input 1 polarity
/io/in/LCL/1/col	I	1..18		Local Input 1 color
/io/in/LCL/1/name	S		16 chars max	Local Input 1 name
/io/in/LCL/1/icon	I	0..999		Local Input 1 icon (indexed)
/io/in/LCL/1/tags	S		80 chars max	Local Input 1 tags
/io/in/LCL/1/$ha	I	0..5		Local input 1 ha type [RO]
/io/in/LCL/1/rmt	S		OFF, AES A, AES B, AES C	Local input 1 remote control
/io/in/LCL/1/$ract	I	0..1		Local input 1 remote active [RO]
/io/in/LCL/1/$rdest	S		7 chars max	Local input 1 remote dest [RO]
/io/in/LCL/1/rcvc	I	0..1		Local input 1 remote customizations sync
/io/in/LCL/1/$mute	I	0..2		Local input 1 mute [RO]
/io/in/AUX	N			Aux Input node
/io/in/AUX/1	N	1..8		Aux Input 1 node
/io/in/AUX/1/mode	S		M, ST, M/S	Aux Input 1 mode
/io/in/AUX/1/mute	I	0..1		Aux Input 1 mute
/io/in/AUX/1/pol	I	0..1		Aux Input 1 polarity
/io/in/AUX/1/col	I	1..18		Aux Input 1 color
/io/in/AUX/1/name	S		16 chars max	Aux Input 1 name
/io/in/AUX/1/icon	I	0..999		Aux Input 1 icon (indexed)
/io/in/AUX/1/tags	S		80 chars max	Aux Input 1 tags
/io/in/AUX/1/$mute	I	0..2		Aux input 1 mute [RO]
/io/in/A	N			AES50 A Input node
/io/in/A/1	N	1..48		AES50 A Input 1 node
/io/in/A/1/mode	S		M, ST, M/S	AES50 A Input 1 mode
/io/in/A/1/g	F	-3..45.5	98 steps	AES50 A Input 1 gain (dB)
/io/in/A/1/vph	I	0..1		AES50 A Input 1 phantom power
/io/in/A/1/mute	I	0..1		AES50 A Input 1 mute
/io/in/A/1/pol	I	0..1		AES50 A Input 1 polarity
/io/in/A/1/col	I	1..18		AES50 A Input 1 color
/io/in/A/1/name	S		16 chars max	AES50 A Input 1 name
/io/in/A/1/icon	I	0..999		AES50 A Input 1 icon (indexed)
/io/in/A/1/tags	S		80 chars max	AES50 A Input 1 tags

[18] All 24 local inputs may not be available depending on the console model

/io/in/A/1/$ha	I	0..5		AES50 A input 1 ha type [RO]
/io/in/A/1/rmt	S		OFF, AES A, AES B, AES C	AES50 A input 1 remote control
/io/in/A/1/$ract	I	0..1		AES50 A input 1 remote active [RO]
/io/in/A/1/$rdest	S		7 chars max	AES50 A input 1 remote dest [RO]
/io/in/A/1/rcvc	I	0..1		AES50 A input 1 remote customizations sync
/io/in/A/1/$mute	I	0..2		AES50 A input 1 mute [RO]
/io/in/B	N			AES50 B Input node
/io/in/B/1	N	1..48		AES50 B Input 1 node
/io/in/B/1/mode	S		M, ST, M/S	AES50 B Input 1 mode
/io/in/B/1/g	F	-3..45.5	98 steps	AES50 B Input 1 gain (dB)
/io/in/B/1/vph	I	0..1		AES50 B Input 1 phantom power
/io/in/B/1/mute	I	0..1		AES50 B Input 1 mute
/io/in/B/1/pol	I	0..1		AES50 B Input 1 polarity
/io/in/B/1/col	I	1..18		AES50 B Input 1 color
/io/in/B/1/name	S		16 chars max	AES50 B Input 1 name
/io/in/B/1/icon	I	0..999		AES50 B Input 1 icon (indexed)
/io/in/B/1/tags	S		80 chars max	AES50 B Input 1 tags
/io/in/B/1/$ha	I	0..5		AES50 B input 1 ha type [RO]
/io/in/B/1/rmt	S		OFF, AES A, AES B, AES C	AES50 B input 1 remote control
/io/in/B/1/$ract	I	0..1		AES50 B input 1 remote active [RO]
/io/in/B/1/$rdest	S		7 chars max	AES50 B input 1 remote dest [RO]
/io/in/B/1/rcvc	I	0..1		AES50 B input 1 remote customizations sync
/io/in/B/1/$mute	I	0..2		AES50 B input 1 mute [RO]
/io/in/C	N			AES50 C Input node
/io/in/C/1	N	1..48		AES50 C Input 1 node
/io/in/C/1/mode	S		M, ST, M/S	AES50 C Input 1 mode
/io/in/C/1/g	F	-3..45.5	98 steps	AES50 C Input 1 gain (dB)
/io/in/C/1/vph	I	0..1		AES50 C Input 1 phantom power
/io/in/C/1/mute	I	0..1		AES50 C Input 1 mute
/io/in/C/1/pol	I	0..1		AES50 C Input 1 polarity
/io/in/C/1/col	I	1..18		AES50 C Input 1 color
/io/in/C/1/name	S		16 chars max	AES50 C Input 1 name
/io/in/C/1/icon	I	0..999		AES50 C Input 1 icon (indexed)
/io/in/C/1/tags	S		80 chars max	AES50 C Input 1 tags
/io/in/C/1/$ha	I	0..4		AES50 C input 1 ha type [RO]
/io/in/C/1/rmt	S		OFF, AES A, AES B, AES C	AES50 C input 1 remote control
/io/in/C/1/$ract	I	0..1		AES50 C input 1 remote active [RO]
/io/in/C/1/$rdest	S		7 chars max	AES50 C input 1 remote dest [RO]
/io/in/C/1/rcvc	I	0..1		AES50 C input 1 remote customizations sync
/io/in/C/1/$mute	I	0..2		AES50 C input 1 mute [RO]
/io/in/SC	N			StageConnect Input node
/io/in/SC/1	N	1..32		StageConnect Input 1 node
/io/in/SC/1/mode	S		M, ST, M/S	StageConnect Input 1 mode
/io/in/SC/1/mute	I	0..1		StageConnect Input 1 mute
/io/in/SC/1/pol	I	0..1		StageConnect Input 1 polarity
/io/in/SC/1/col	I	1..18		StageConnect Input 1 color

/io/in/SC/1/name	S		16 chars max	StageConnect Input 1 name
/io/in/SC/1/icon	I	0..999		StageConnect Input 1 icon (indexed)
/io/in/SC/1/tags	S		80 chars max	StageConnect Input 1 tags
/io/in/SC/1/$mute	I	0..2		StageConnect 1 mute [RO]
/io/in/USB	N			USB Input node
/io/in/USB/1	N	1..48		USB Input 1 node
/io/in/USB/1/mode	S		M, ST, M/S	USB Input 1 mode
/io/in/USB/1/mute	I	0..1		USB Input 1 mute
/io/in/USB/1/pol	I	0..1		USB Input 1 polarity
/io/in/USB/1/col	I	1..18		USB Input 1 color
/io/in/USB/1/name	S		16 chars max	USB Input 1 name
/io/in/USB/1/icon	I	0..999		USB Input 1 icon (indexed)
/io/in/USB/1/tags	S		80 chars max	USB Input 1 tags
/io/in/USB/1/$mute	I	0..2		USB Input 1 mute [RO]
/io/in/CRD	N			Card Input node
/io/in/CRD/1	N	1..64		Card Input 1 node
/io/in/CRD/1/mode	S		M, ST, M/S	Card Input 1 mode
/io/in/CRD/1/mute	I	0..1		Card Input 1 mute
/io/in/CRD/1/pol	I	0..1		Card Input 1 polarity
/io/in/CRD/1/col	I	1..18		Card Input 1 color
/io/in/CRD/1/name	S		16 chars max	Card Input 1 name
/io/in/CRD/1/icon	I	0..999		Card Input 1 icon (indexed)
/io/in/CRD/1/tags	S		80 chars max	Card Input 1 tags
/io/in/CRD/1/$mute	I	0..2		Card Input 1 mute [RO]
/io/in/MOD	N			Module Input node
/io/in/MOD/1	N	1..64		Module Input 1 node
/io/in/MOD/1/mode	S		M, ST, M/S	Module Input 1 node
/io/in/MOD/1/mute	I	0..1		Module Input 1 mute
/io/in/MOD/1/pol	I	0..1		Module Input 1 polarity
/io/in/MOD/1/col	I	1..18		Module Input 1 color
/io/in/MOD/1/name	S		16 chars max	Module Input 1 name
/io/in/MOD/1/icon	I	0..999		Module Input 1 icon (indexed)
/io/in/MOD/1/tags	S		80 chars max	Module Input 1 tags
/io/in/MOD/1/$mute	I	0..2		Module Input 1 mute [RO]
/io/in/PLAY	N			USB Player Input node
/io/in/PLAY/1	N	1..4		USB Player Input 1 node
/io/in/PLAY/1/mode	S		M, ST, M/S	USB Player Input 1 mode
/io/in/PLAY/1/mute	I	0..1		USB Player Input 1 mute
/io/in/PLAY/1/pol	I	0..1		USB Player Input 1 polarity
/io/in/PLAY/1/col	I	1..18		USB Player Input 1 color
/io/in/PLAY/1/name	S		16 chars max	USB Player Input 1 name
/io/in/PLAY/1/icon	I	0..999		USB Player Input 1 icon (indexed)
/io/in/PLAY/1/tags	S		80 chars max	USB Player Input 1 tags
/io/in/PLAY/1/$mute	I	0..2		USB Player Input 1 mute [RO]
/io/in/AES	N			AES/EBU Input node
/io/in/AES/1	N	1..2		AES/EBU Input 1 node
/io/in/AES/1/mode	S		M, ST, M/S	AES/EBU Input 1 mode

/io/in/AES/1/mute	I	0..1		AES/EBU Input 1 mute
/io/in/AES/1/pol	I	0..1		AES/EBU Input 1 polarity
/io/in/AES/1/col	I	1..18		AES/EBU Input 1 color
/io/in/AES/1/name	S		16 chars max	AES/EBU Input 1 name
/io/in/AES/1/icon	I	0..999		AES/EBU Input 1 icon (indexed)
/io/in/AES/1/tags	S		80 chars max	AES/EBU Input 1 tags
/io/in/AES/1/$mute	I	0..2		AES/EBU Input 1 mute [RO]
/io/in/USR	N			User Signal Input node
/io/in/USR/1	N	1..56		User Signal Input 1 node: 1-24 are User Signals, 25-56 are User Patches
/io/in/USR/1/mode	S		M, ST, M/S	User Signal Input 1 mode
/io/in/USR/1/mute	I	0..1		User Signal Input 1 mute
/io/in/USR/1/pol	I	0..1		User Signal Input 1 polarity
/io/in/USR/1/col	I	1..18		User Signal Input 1 color
/io/in/USR/1/name	S		16 chars max	User Signal Input 1 name
/io/in/USR/1/icon	I	0..999		User Signal Input 1 icon (indexed)
/io/in/USR/1/tags	S		80 chars max	User Signal Input 1 tags
/io/in/USR/1/$mute	I	0..2		User Signal Input 1 mute [RO]
/io/in/USR/1/user	N	1..56		User Signal 1..24 source node User Patch 25..56 source node
/io/in/USR/1/user/grp	S S		OFF, CH, AUX, BUS, MAIN, MTX OFF, LCL, AUX, A, B, C, SC, USB, CRD, MOD, PLAY, AES	User Signal source group User Patch source group
/io/in/USR/1/user/in	I I	1..40 1..64		User Signal source number User Patch source number
/io/in/USR/1/user/tap[19]	S		PRE, POST	User Signal source tap point
/io/in/USR/1/user/lr[20]	S		L+R, L, R	User Signal source take
/io/in/OSC	N			Oscillator Input node
/io/in/OSC/1	N	1..2		Oscillator Input 1 node
/io/in/OSC/1/mode	S		M, ST, M/S	Oscillator Input 1 mode [RO]
/io/in/OSC/1/mute	I	0..1		Oscillator Input 1 mute
/io/in/OSC/1/col	I	1..18		Oscillator Input 1 color
/io/in/OSC/1/name	S		16 chars max	Oscillator Input 1 name
/io/in/OSC/1/icon	I	0..999		Oscillator Input 1 icon (indexed)
/io/in/OSC/1/tags	S		80 chars max	Oscillator Input 1 tags
/io/in/OSC/1/$mute	I	0..2		Oscillator Input 1 mute [RO]
/io/in/OSC/1/osc	N			Oscillator 1 source node
/io/in/OSC/1/osc/lvl	F	-40..6	69 steps	Oscillator 1 source level
/io/in/OSC/1/osc/mode	S		SINE, PINK, WHITE	Oscillator 1 source mode
/io/in/OSC/1/osc/f	F	20…20000	2323 steps	Oscillator 1 source frequency
/io/in/$BUS	N			Bus Input node
/io/in/$BUS/1	N	1..32		Bus Input 1 node
/io/in/$BUS/1/mode	S		M, ST, M/S	Bus Input 1 mode [RO]

[19] Only for nodes 1..24 [User Signals]

[20] Only for nodes 1..24 [User Signals]

/io/in/$BUS/1/col	I	1..18		Bus Input 1 color [RO]
/io/in/$BUS/1/name	S		16 chars max	Bus Input 1 name [RO]
/io/in/$BUS/1/icon	I	0..999		Bus Input 1 icon [RO]
/io/in/$BUS/1/tags	S		80 chars max	Bus Input 1 tags [RO]
/io/in/$MAIN	N			Main Input node
/io/in/$MAIN/1	N	1..8		Main Input 1 node
/io/in/$MAIN/1/mode	S		M, ST, M/S	Main Input 1 mode [RO]
/io/in/$MAIN/1/col	I	1..18		Main Input 1 color [RO]
/io/in/$MAIN/1/name	S		16 chars max	Main Input 1 name [RO]
/io/in/$MAIN/1/icon	I	0..999		Main Input 1 icon [RO]
/io/in/$MAIN/1/tags	S		80 chars max	Main Input 1 tags [RO]
/io/in/$MTX	N			Matrix Input node
/io/in/$MTX/1	N	1..16		Matrix Input 1 node
/io/in/$MTX/1/mode	S		M, ST, M/S	Matrix Input 1 mode [RO]
/io/in/$MTX/1/col	I	1..18		Matrix Input 1 color [RO]
/io/in/$MTX/1/name	S		16 chars max	Matrix Input 1 name [RO]
/io/in/$MTX/1/icon	I	0..999		Matrix Input 1 icon [RO]
/io/in/$MTX/1/tags	S		80 chars max	Matrix Input 1 tags [RO]
/io/in/$SEND	N			FX Send Input node
/io/in/$SEND/1	N	1..32		FX Send Input 1 node
/io/in/$SEND/1/mode	S		M, ST, M/S	FX Send Input 1 mode [RO]
/io/in/$SEND/1/col	I	1..18		FX Send Input 1 color [RO]
/io/in/$SEND/1/name	S		16 chars max	FX Send Input 1 name [RO]
/io/in/$SEND/1/icon	I	0..999		FX Send Input 1 icon [RO]
/io/in/$SEND/1/tags	S		80 chars max	FX Send Input 1 tags [RO]
/io/in/$MON	N			Monitor Input node
/io/in/$MON/1	N	1..4		Monitor Input 1 node
/io/in/$MON/1/mode	S		M, ST, M/S	Monitor Input 1 mode [RO]
/io/in/$MON/1/col	I	1..18		Monitor Input 1 color [RO]
/io/in/$MON/1/name	S		16 chars max	Monitor Input 1 name [RO]
/io/in/$MON/1/icon	I	0..999		Monitor Input 1 icon [RO]
/io/in/$MON/1/tags	S		80 chars max	Monitor Input 1 tags [RO]
/io/out	N			Output node
/io/out/LCL	N			Local Output node
/io/out/LCL/1	N	1..8		Local Output 1 node
/io/out/LCL/1/grp	S		OFF, LCL, AUX, A, B, C, SC, USB, CRD, MOD, PLAY, AES, USR, OSC, BUS, MAIN, MTX, SEND, MON	Local Output 1 group
/io/out/LCL/1/in	I	1..64		Local Output 1 input
/io/out/AUX	N			Aux Output node
/io/out/AUX/1	N	1..8[21]		Aux Output 1 node
/io/out/AUX/1/grp	S		OFF, LCL, AUX, A, B, C, SC, USB, CRD, MOD, PLAY, AES, USR, OSC, BUS, MAIN, MTX, SEND, MON	Aux Output 1 group

[21] Aux 1..6 may be 'not available' on some models

/io/out/AUX/1/in	I	1..64		Aux Output 1 input
/io/out/A	N			AES50 A Output node
/io/out/A/1	N	1..48		AES50 A Output 1 node
/io/out/A/1/grp	S		OFF, LCL, AUX, A, B, C, SC, USB, CRD, MOD, PLAY, AES, USR, OSC, BUS, MAIN, MTX, SEND, MON	AES50 A Output 1 group
/io/out/A/1/in	I	1..64		AES50 A Output 1 input
/io/out/B	N			AES50 B Output node
/io/out/B/1	N	1..48		AES50 B Output 1 node
/io/out/B/1/grp	S		OFF, LCL, AUX, A, B, C, SC, USB, CRD, MOD, PLAY, AES, USR, OSC, BUS, MAIN, MTX, SEND, MON	AES50 B Output 1 group
/io/out/B/1/in	I	1..64		AES50 B Output 1 input
/io/out/C	N			AES50 C Output node
/io/out/C/1	N	1..48		AES50 C Output 1 node
/io/out/C/1/grp	S		OFF, LCL, AUX, A, B, C, SC, USB, CRD, MOD, PLAY, AES, USR, OSC, BUS, MAIN, MTX, SEND, MON	AES50 C Output 1 group
/io/out/C/1/in	I	1..64		AES50 C Output 1 input
/io/out/SC	N			StageConnect Output node
/io/out/SC/1	N	1..32		StageConnect Output 1 node
/io/out/SC/1/grp	S		OFF, LCL, AUX, A, B, C, SC, USB, CRD, MOD, PLAY, AES, USR, OSC, BUS, MAIN, MTX, SEND, MON	StageConnect Output 1 group
/io/out/SC/1/in	I	1..64		StageConnect Output 1 input
/io/out/USB	N			USB Output Audio node
/io/out/USB/1	N	1..48		USB Output Audio 1 node
/io/out/USB/1/grp	S		OFF, LCL, AUX, A, B, C, SC, USB, CRD, MOD, PLAY, AES, USR, OSC, BUS, MAIN, MTX, SEND, MON	USB Output Audio 1 group
/io/out/USB/1/in	I	1..64		USB Output Audio 1 input
/io/out/CRD	N			Card Output node
/io/out/CRD/1	N	1..64		Card Output 1 node
/io/out/CRD/1/grp	S		OFF, LCL, AUX, A, B, C, SC, USB, CRD, MOD, PLAY, AES, USR, OSC, BUS, MAIN, MTX, SEND, MON	Card Output 1 group
/io/out/CRD/1/in	I	1..64		Card Output 1 input
/io/out/MOD	N			Module Output node
/io/out/MOD/1	N	1..64		Module Output 1 node
/io/out/MOD/1/grp	S		OFF, LCL, AUX, A, B, C, SC, USB, CRD, MOD, PLAY, AES, USR, OSC, BUS, MAIN, MTX, SEND, MON	Module Output 1 group
/io/out/MOD/1/in	I	1..64		Module Output 1 input
/io/out/REC	N			USB Record Output node

/io/out/REC/1	N	1..4		USB Record Output 1 node
/io/out/REC/1/grp	S		OFF, LCL, AUX, A, B, C, SC, USB, CRD, MOD, PLAY, AES, USR, OSC, BUS, MAIN, MTX, SEND, MON	USB Record Output 1 group
/io/out/REC/1/in	I	1..64		USB Record Output 1 input
/io/out/AES	N			AES/EBU Output node
/io/out/AES/1	N	1..2		AES/EBU Output 1 node
/io/out/AES/1/grp	S		OFF, LCL, AUX, A, B, C, SC, USB, CRD, MOD, PLAY, AES, USR, OSC, BUS, MAIN, MTX, SEND, MON	AES/EBU Output 1 group
/io/out/AES/1/in	I	1..64		AES/EBU Output 1 input

Channel Settings

Command	Type	Range	Text	Description
/ch	N			Channel node
/ch/1	N	1..40		Channel 1 node
/ch/1/in	N			Channel 1 input node
/ch/1/in/set	N			Channel 1 input set node
/ch/1/in/set/$mode	S		M, ST, M/S	Channel 1 input mode [RO]
/ch/1/in/set/srcauto	I	0..1		Channel 1 input auto source switch
/ch/1/in/set/altsrc	I	0..1		Channel 1 input main/alt switch
/ch/1/in/set/inv	I	0..1		Channel 1 input phase invert switch
/ch/1/in/set/trim	F	-18..18	361 steps	Channel 1 input trim (dB)
/ch/1/in/set/bal	F	-9..9	181 steps	Channel 1 input balance (dB)
/ch/1/in/set/$g	F	-2.5..45 -3.0..45.5	20 steps (LCL) 98 steps (AES)	Channel 1 input gain (dB) – depends on source type
/ch/1/in/set/$vph	I	0..1		Channel 1 input phantom power – depends on source type
/ch/1/in/set/dlymode	S		M, FT, MS, SMP	Meters, feet, milliseconds, samples
/ch/1/in/set/dly	F	0..150 m / 0.5..500 ft / 0.5..500 ms / 16..500 smp	1501 steps / 1000 steps / 4996 steps / 485 steps	Channel 1 input delay (meters, feet, ms, samples)
/ch/1/in/set/dlyon	I	0..1		Channel 1 input delay
/ch/1/in/conn	N			Channel 1 input connection node
/ch/1/in/conn/grp	S		OFF, LCL, AUX, A, B, C, SC, USB, CRD, MOD, PLAY, AES, USR, OSC, BUS, MAIN, MTX	Channel 1 main input connection group
/ch/1/in/conn/in	I	1..64		Channel 1 main input connection group index
/ch/1/in/conn/altgrp	S		OFF, LCL, AUX, A, B, C, SC, USB, CRD, MOD, PLAY, AES, USR, OSC, BUS, MAIN, MTX	Channel 1 alt input connection group
/ch/1/in/conn/altin	I	1..64		Channel 1 alt input connection group index
/ch/1/flt	N			Channel 1 filter node
/ch/1/flt/lc	I	0..1		Channel 1 low cut switch
/ch/1/flt/lcf	F	20..2000	641 steps	Channel 1 low cut frequency (Hz)
/ch/1/flt/lcs	S		6, 12, 18, 24	Channel 1 low cut slope
/ch/1/flt/hc	I	0..1		Channel 1 high cut switch
/ch/1/flt/hcf	F	50..20000	833 steps	Channel 1 high cut frequency (Hz)
/ch/1/flt/hcs	S		6, 12	Channel 1 high cut slope
/ch/1/flt/tf	I	0..1		Channel 1 tool filter switch
/ch/1/flt/mdl	S		TILT, MAX, AP1, AP2	Channel 1 filter model (see Appendix on Filter plugins for parameters details, OSC patterns in italic below correspond to TILT)
/ch/1/flt/tilt[22]	F	-6..6	49 steps	Channel 1 tilt level (dB)

[22] This is for the TILT filter model. Can use more parameters depending on model type

/ch/1/clink	I	0..1		Channel 1 custom link
/ch/1/col	I	1..18		Channel 1 color
/ch/1/name	S		16 chars max	Channel 1 name
/ch/1/icon	I	0..999		Channel 1 icon
/ch/1/led	I	0..1		Channel 1 scribble light
/ch/1/$col	I	1..18		Channel 1 color [RO] reflects linked source or current strip value
/ch/1/$name	S		16 chars max	Channel 1 name [RO] reflects linked source or current strip value
/ch/1/$icon	I	0..999		Channel 1 icon [RO] reflects linked source or current strip value
/ch/1/mute	I	0..1		Channel 1 mute
/ch/1/fdr	F	-144..10	-oo..10 in 1024 steps	Channel 1 fader
/ch/1/pan	F	-100..100	201 steps	Channel 1 pan
/ch/1/wid	F	-150..150	61 steps	Channel 1 width (%)
/ch/1/$solo	I	0..1		Channel 1 solo switch
/ch/1/$sololed	I	0..2		Channel 1 solo LED [RO]
/ch/1/solosafe	I	0..1		Channel 1 solo safe
/ch/1/mon	S		A, B, A+B	Channel 1 monitor mode
/ch/1/proc	S		GEDI, GEID, GIED, IGED, GDEI, GDIE, GIDE, IGDE, EGDI, EGID, EIGD, IEGD, EDGI, EDIG, EIDG, IEDG, DEGI, DEIG, DIEG, IDEG, DGEI, DGIE, DIGE, IDGE	Channel 1 process order (G: Gate, E: EQ, D: Dynamics, I: Insert
/ch/1/ptap	S		IN, FILT, 3, 4, 5, PFL, AFL, POST	Channel 1 pretap (to sends)
/ch/1/$presolo	I	0..1		Channel 1 presolo
/ch/1/peq	N			Channel 1 PreSend EQ node
/ch/1/peq/on	I	0..1		Channel 1 PEQ switch
/ch/1/peq/1g	F	-15..15	301 steps	Channel 1 PEQ band 1 gain (dB)
/ch/1/peq/1f	F	20..20000	960 steps	Channel 1 PEQ band 1 frequency (Hz)
/ch/1/peq/1q	F	0.44..10	181 steps	Channel 1 PEQ band 1 Q
/ch/1/peq/2g	F	-15..15	301 steps	Channel 1 PEQ band 2 gain 9dB)
/ch/1/peq/2f	F	20..20000	960 steps	Channel 1 PEQ band 2 frequency (Hz)
/ch/1/peq/2q	F	0.44..10	181 steps	Channel 1 PEQ band 2 Q
/ch/1/peq/3g	F	-15..15	301 steps	Channel 1 PEQ band 3 gain 9dB)
/ch/1/peq/3f	F	20..20000	960 steps	Channel 1 PEQ band 3 frequency (Hz)
/ch/1/peq/3q	F	0.44..10	181 steps	Channel 1 PEQ band 3 Q
/ch/1/gate	N			Channel 1 gate node
/ch/1/gate/on	I	0..1		Channel 1 gate switch
/ch/1/gate/mdl	S		GATE, DUCK, E88, 9000G, D241G, DS902, DEQ, DEQ2, WAVE, PSE, CMB, RIDE, WARM, COMP, EXP, B160, B560, D241C, ECL33, 9000C, SBUS, RED3, 76LA, LA, F670,	Channel 1 gate model (see Appendix on Gate plugins for parameters details, OSC patterns in italic below correspond to GATE)

			BLISS, NSTR, 2250, L100, E88C, LMT, ONEC[23]	
/ch/1/gate/mix	F	0..100	101 steps	Channel 1 gate mix (%)
/ch/1/gate/gain	F	-6..12	37 steps	Channel 1 gate gain (dB)
/ch/1/gate/thr[24]	F	-80..0	161 steps	Channel 1 gate threshold (dB)
/ch/1/gate/range	F	3..60	115 steps	Channel 1 gate range (dB)
/ch/1/gate/att	F	0..120	121 steps	Channel 1 gate attack (ms)
/ch/1/gate/hld	F	0..200	200 steps	Channel 1 gate hold (ms)
/ch/1/gate/rel	F	4..4000	130 steps	Channel 1 gate release(ms)
/ch/1/gate/acc	F	0..100	21 steps	Channel 1 gate accent (5)
/ch/1/gate/ratio	S		1:1.5, 1:2, 1:3, 1:4, gate	Channel 1 gate ratio
/ch/1/gatesc	N			Channel 1 gate sidechain node
/ch/1/gatesc/type	S		Off, LP12, HP12, BP, NOTCH	Channel 1 gate sidechain type
/ch/1/gatesc/f	F	20..20000	961 steps	Channel 1 gate sidechain frequency (Hz)
/ch/1/gatesc/q	F	0.44..10	181 steps	Channel 1 gate sidechain Q
/ch/1/gatesc/src	S		SHELF, CH.1..CH.40	Channel 1 gate sidechain source
/ch/1/gatesc/tap	S		IN, FILT, 3, 4, 5, PFL, AFL, POST	Channel 1 gate sidechain tap
/ch/1/gatesc/$solo	I	0..1		Channel 1 gate sidechain solo
/ch/1/eq	N			Channel 1 EQ node
/ch/1/eq/on	I	0..1		Channel 1 EQ switch
/ch/1/eq/mdl	S		STD, SOUL, E88, E84, F110, PULSAR, MACH4	Channel 1 EQ model (see Appendix on EQ plugins for parameters details, OSC patterns in italic below correspond to STD)
/ch/1/eq/mix	F	0..125	126 steps	Channel 1 EQ mix (%)
/ch/1/eq/$solo	I	0..1		Channel 1 EQ solo
/ch/1/eq/$solobd	I	0..6		Channel 1 EQ solo band
/ch/1/eq/lg[25]	F	-15..15	301 steps	Channel 1 EQ low gain (dB)
/ch/1/eq/lf	F	20..2000	641 steps	Channel 1 EQ low frequency (Hz)
/ch/1/eq/lq	F	0.44..10	181 steps	Channel 1 EQ low Q
/ch/1/eq/leq	S		PEQ, SHV	Channel 1 EQ low type
/ch/1/eq/1g	F	-15..15	301 steps	Channel 1 EQ band 1 gain (dB)
/ch/1/eq/1f	F	20..20000	961 steps	Channel 1 EQ band 1 frequency (Hz)
/ch/1/eq/1q	F	0.44..10	181 steps	Channel 1 EQ band 1 Q
/ch/1/eq/2g	F	-15..15	301 steps	Channel 1 EQ band 2 gain (dB)
/ch/1/eq/2f	F	20..20000	961 steps	Channel 1 EQ band 2 frequency (Hz)
/ch/1/eq/2q	F	0.44..10	181 steps	Channel 1 EQ band 2 Q
/ch/1/eq/3g	F	-15..15	301 steps	Channel 1 EQ band 3 gain (dB)
/ch/1/eq/3f	F	20..20000	961 steps	Channel 1 EQ band 3 frequency (Hz)
/ch/1/eq/3q	F	0.44..10	181 steps	Channel 1 EQ band 3 Q
/ch/1/eq/4g	F	-15..15	301 steps	Channel 1 EQ band 4 gain (dB)
/ch/1/eq/4f	F	20..20000	961 steps	Channel 1 EQ band 4 frequency (Hz)
/ch/1/eq/4q	F	0.44..10	181 steps	Channel 1 EQ band 4 Q
/ch/1/eq/hg	F	-15..15	301 steps	Channel 1 EQ high gain (dB)
/ch/1/eq/hf	F	50..20000	833 steps	Channel 1 EQ high frequency (Hz)

[23] Side chain EQ is not available when auxcombo [CMB] is used in dyn slot

[24] This is for the GATE gate model. Can use more parameters depending on model type

[25] This is for the STD eq model. Can use more parameters depending on model type

/ch/1/eq/hq	F	0.44..10	181 steps	Channel 1 EQ high Q
/ch/1/eq/heq	S		PEQ, SHV	Channel 1 EQ high type
/ch/1/dyn	N			Channel 1 dynamic (compressor) node
/ch/1/dyn/on	I	0..1		Channel 1 compressor switch
/ch/1/dyn/mdl	S		GATE, DUCK, E88, 9000G, D241G, DS902, DEQ, DEQ2, WAVE, PSE, CMB, RIDE, WARM, COMP, EXP, B160, B560, D241C, ECL33, 9000C, SBUS, RED3, 76LA, LA, F670, BLISS, NSTR, 2250, L100, E88C, LMT, ONEC [26]	Channel 1 compressor model (see Appendix on Compressor plugins for parameters details, OSC patterns in italic below correspond to COMP)
/ch/1/dyn/mix	F	0..100	101 steps	Channel 1 compressor mix (%)
/ch/1/dyn/gain	F	-6..12	37 steps	Channel 1 compressor gain (dB)
/ch/1/dyn/thr[27]	F	-60..0	121 steps	Channel 1 compressor threshold (dB)
/ch/1/dyn/ratio	S		1.1, 1.2, 1.3, 1.5, 1.7, 2.0, 2.5, 3.0, 3.5, 4.0, 5.0, 6.0, 8.0, 10, 20, 50, 100	Channel 1 compressor ratio
/ch/1/dyn/knee	I	0..5		Channel 1 compressor knee
/ch/1/dyn/det	S		PEAK, RMS	Channel 1 compressor detect
/ch/1/dyn/att	F	0..120	121 steps	Channel 1 compressor attack (ms)
/ch/1/dyn/hld	F	1..200	200 steps	Channel 1 compressor hold (ms)
/ch/1/dyn/rel	F	4..4000	130 steps	Channel 1 compressor release (ms)
/ch/1/dyn/env	S		LIN, LOG	Channel 1 compressor envelope
/ch/1/dyn/auto	I	0..1		Channel 1 compressor auto switch
/ch/1/dynxo	N			Channel 1 compressor crossover node
/ch/1/dynxo/depth	F	0..20	41 steps	Channel 1 compressor crossover depth (dB)
/ch/1/dynxo/type	S		OFF, LO, HI, S6, S12, PEAK	Channel 1 compressor crossover type
/ch/1/dynxo /f	F	20..20000	901 steps	Channel 1 compressor crossover frequency (Hz)
/ch/1/dynxo/q	F	0.40..4.00	121 steps	Channel 1 compressor crossover quality factor
/ch/1/dynxo/$solo	I	0..1		Channel 1 compressor crossover solo
/ch/1/dynsc	N			Channel 1 compressor sidechain node
/ch/1/dynsc/type	S		OFF, LP12, HP12, BP, NOTCH	Channel 1 compressor sidechain type
/ch/1/dynsc/f	F	20..20000	901 steps	Channel 1 compressor sidechain frequency (Hz)
/ch/1/dynsc/q	F	0.44..10	181 steps	Channel 1 compressor sidechain Q
/ch/1/dynsc/src	S		SELF, CH.1..CH.40	Channel 1 compressor sidechain source
/ch/1/dynsc/tap	S		IN, FILT, 3, 4, 5, PFL, AFL, POST	Channel 1 compressor sidechain tap
/ch/1/dynsc/$solo	I	0..1		Channel 1 compressor sidechain solo
/ch/1/preins	N			Channel 1 pre-insert node
/ch/1/preins/on	I	0..1		Channel 1 pre-insert switch

[26] Side chain EQ is not available when auxcombo [CMB] is used in dyn slot

[27] This is for the COMP dyn model. Can use more parameters depending on model type

/ch/1/preins/ins	S		NONE, FX1..FX16	Channel 1 pre-insert FX slot
/ch/1/preins/$stat	S		-, OK, N/A	Channel 1 pre-insert status [RO]
/ch/1/main	N			Channel 1 Main node
/ch/1/main/1	N	1..4[28]		Channel 1 Main 1 node
/ch/1/main/1/on	I	0..1		Channel 1 Main 1 on switch
/ch/1/main/1/lvl	F	-144..10	-oo..10 in 1024 steps	Channel 1 Main 1 fader level (dB)
/ch/1/main/1/pre	I	0..1		Channel 1 sent pre fader to Main 1
/ch/1/send	N			Channel 1 sends node
/ch/1/send/1	N	1..16		Channel 1 sends 1 node
/ch/1/send/1/on	I	0..1		Channel 1 sends 1 on switch
/ch/1/send/1/lvl	F	-144..10	-oo..10 in 1024 steps	Channel 1 sends 1 fader level (dB)
/ch/1/send/1/pon	I	0..1		Channel 1 sends 1 pre always on switch
/ch/1/send/1/mode	S		PRE, POST, GRP	Channel 1 sends 1 mode
/ch/1/send/1/plink	I	0..2		Channel 1 sends 1 pan link (0=individual)
/ch/1/send/1/pan	F	-100..100	201 steps	Channel 1 sends 1 pan
/ch/1/send/MX<x>	N	<x>:1..8		Channel 1 sends matrix <x> node
/ch/1/send/MX<x>/on	I	0..1		Channel 1 sends mtx on switch
/ch/1/send/MX<x>/lvl	F	-144..10	-oo..10 in 1024 steps	Channel 1 sends mtx fader level (dB)
/ch/1/send/MX<x>/pon	I	0..1		Channel 1 sends mtx pre always on switch (ignore channel mute)
/ch/1/send/MX<x>/mode	S		PRE, POST, GRP	Channel 1 sends mtx mode
/ch/1/send/MX<x>/plink	I	0..2		Channel 1 sends mtx pan link (0=individual)
/ch/1/send/MX<x>/pan[29]	F	-100..100	201 steps	Channel 1 sends mtx pan
/ch/1/tapwid	F	-150..150		Channel 1 width
/ch/1/postins	N			Channel 1 post insert node
/ch/1/postins/on	I	0..1		Channel 1 post insert on switch
/ch/1/postins/mode	S		FX, AUTO_X, AUTO_Y	Channel 1 post insert mode
/ch/1/postins/ins	S		NONE, FX1..FX16	Channel 1 post insert FX slot
/ch/1/postins/w	F	-12..12	241 steps	Channel 1 post insert autogain weight
/ch/1/postins/$stat	S		-, OK, N/A	Channel 1 post insert status [RO]
/ch/1/tags	S		80 chars max	Channel 1 tags[30]
/ch/1/$fdr	F	-144..10	-oo..10 in 1024 steps	Channel 1 fader level as affected by dca (dB) [RO]
/ch/1/$mute	I	0..2		Channel 1 mute [RO]
/ch/1/$muteovr	I	0..1		Channel 1 mute override

[28] 1..8 on WING Rack model, used for the 4 headphones stereo outputs on the back on the rack.

[29] plink must be 0 for this parameter to be settable

[30] Tags #D1..#D16 are 'reserved' for DCA1..16 assignment

Aux Settings

Command	Type	Range	Text	Description
/aux	N			Aux node
/aux/1	N	1..8		Aux 1 node
/aux/1/in	N			Aux 1 input node
/aux/1/in/set	N			Aux 1 input set node
/aux/1/in/set/$mode	S		M, ST, M/S	Aux 1 input mode [RO]
/aux/1/in/set/srcauto	I	0..1		Aux 1 input auto source switch
/aux/1/in/set/altsrc	I	0..1		Aux 1 input main/alt switch
/aux/1/in/set/inv	I	0..1		Aux 1 input phase invert switch
/aux/1/in/set/trim	F	-18..18	361 steps	Aux 1 input trim (dB)
/aux/1/in/set/bal	F	-9..9	181 steps	Aux 1 input balance (dB)
/aux/1/in/set/$g	F	-3..45	98 steps	Aux 1 input gain (dB)
/aux/1/in/set/$vph	I	0..1		Aux 1 input phantom power
/aux/1/in/set/dlymode	S		M, FT, MS, SMP	Meters, feet, milliseconds, samples
/aux/1/in/set/dly	F	0..150 m / 0.5..500 ft / 0.5..500 ms / 16..500 smp	1501 steps / 1000 steps / 4996 steps / 485 steps	Aux 1 input delay (meters, feet, ms, samples)
/aux/1/in/set/dlyon	I	0..1		Aux 1 input delay
/aux/1/in/conn	N			Aux 1 input connection node
/aux/1/in/conn/grp	S		OFF, LCL, AUX, A, B, C, SC, USB, CRD, MOD, PLAY, AES, USR, OSC, BUS, MAIN, MTX	Aux 1 input connection group
/aux/1/in/conn/in	I	1..64		Aux 1 input connection group index
/aux/1/in/conn/altgrp	S		OFF, LCL, AUX, A, B, C, SC, USB, CRD, MOD, PLAY, AES, USR, OSC, BUS, MAIN, MTX	Aux 1 alt input connection group
/aux/1/in/conn/altin	I	1..64		Aux 1 alt input connection group index
/aux/1/clink	I	0..1		Aux 1 custom link
/aux/1/col	I	1..18		Aux 1 color
/aux/1/name	S		16 chars max	Aux 1 name
/aux/1/icon	I	0..999		Aux 1 icon
/aux/1/led	I	0..1		Aux 1 scribble light
/aux/1/$col	I	1..18		Aux 1 color [RO] reflects linked source or current strip value
/aux/1/$name	S		16 chars max	Aux 1 name [RO] reflects linked source or current strip value
/aux/1/$icon	I	0..999		Aux 1 icon [RO] reflects linked source or current strip value
/aux/1/mute	I	0..1		Aux 1 mute
/aux/1/fdr	F	-144..10	-oo..10 in 1024 steps	Aux 1 fader level (dB)
/aux/1/pan	F	-100..100	201 steps	Aux 1 pan
/aux/1/wid	F	-150..150	61 steps	Aux 1 width (%)
/aux/1/$solo	I	0..1		Aux 1 solo
/aux/1/$sololed	I	0..2		Aux 1 solo LED [RO]
/aux/1/solosafe	I	0..1		Aux 1 solo safe
/aux/1/mon	S		A, B, A+B	Aux 1 monitor mode

/aux/1/eq	N			Aux 1 EQ node
/aux/1/eq/on	I	0..1		Aux 1 EQ switch
/aux/1/eq/mdl	S		STD, SOUL, E88, E84, F110, PULSAR	Aux 1 EQ model (see Appendix on EQ plugins for parameters details, OSC patterns in italic below correspond to STD)
/ aux /1/eq/mix	F	0..125	126 steps	Aux 1 EQ mix (%)
/ aux /1/eq/$solo	I	0..1		Aux 1 EQ solo
/ aux /1/eq/$solobd	I	0..6		Aux 1 EQ solo band
/ aux */1/eq/lg*[31]	F	-15..15	301 steps	Aux 1 EQ low gain (dB)
/ aux */1/eq/lf*	F	20..2000	641 steps	Aux 1 EQ low frequency (Hz)
/ aux */1/eq/lq*	F	0.44..10	181 steps	Aux 1 EQ low Q
/ aux */1/eq/leq*	S		PEQ, SHV, CUT	Aux 1 EQ low type
/ aux */1/eq/1g*	F	-15..15	301 steps	Aux 1 EQ band 1 gain (dB)
/ aux */1/eq/1f*	F	20..20000	961 steps	Aux 1 EQ band 1 frequency (Hz)
/ aux */1/eq/1q*	F	0.44..10	181 steps	Aux 1 EQ band 1 Q
/ aux */1/eq/2g*	F	-15..15	301 steps	Aux 1 EQ band 2 gain (dB)
/ aux */1/eq/2f*	F	20..20000	961 steps	Aux 1 EQ band 2 frequency (Hz)
/ aux */1/eq/2q*	F	0.44..10	181 steps	Aux 1 EQ band 2 Q
/ aux */1/eq/3g*	F	-15..15	301 steps	Aux 1 EQ band 3 gain (dB)
/ aux */1/eq/3f*	F	20..20000	961 steps	Aux 1 EQ band 3 frequency (Hz)
/ aux */1/eq/3q*	F	0.44..10	181 steps	Aux 1 EQ band 3 Q
/ aux */1/eq/4g*	F	-15..15	301 steps	Aux 1 EQ band 4 gain (dB)
/ aux */1/eq/4f*	F	20..20000	961 steps	Aux 1 EQ band 4 frequency (Hz)
/ aux */1/eq/4q*	F	0.44..10	181 steps	Aux 1 EQ band 4 Q
/ aux */1/eq/hg*	F	-15..15	301 steps	Aux 1 EQ high gain (dB)
/ aux */1/eq/hf*	F	50..20000	833 steps	Aux 1 EQ high frequency (Hz)
/ aux */1/eq/hq*	F	0.44..10	181 steps	Aux 1 EQ high Q
/ aux */1/eq/heq*	S		PEQ, SHV, CUT	Aux 1 EQ high type
/aux/1/dyn	N			Aux 1 dynamic (PSE/LA combo) node
/aux/1/dyn/on	I	0..1		Aux 1 compressor switch
/aux/1/dyn/mdl	S		GATE, DUCK, E88, 9000G, D241G, WAVE, PSE, CMB, RIDE, COMP, EXP, B160, B560, D241C, ECL33, 9000C, SBUS, RED3, 76LA, LA, F670, BLISS, NSTR, 2250, L100, E88C, LMT, ONEC[32]	Aux 1 compressor model (see Appendix on Compressor plugins for parameters details, OSC patterns in italic below correspond to COMP)
/aux/1/dyn/mix	F	0..100	101 steps	Aux 1 compressor mix (%)
/aux/1/dyn/gain	F	-6..12	37 steps	Aux 1 compressor gain (dB)
/aux/1/gate/thr[33]	F	-80..0	161 steps	Aux 1 gate threshold (dB)
/aux/1/gate/range	F	3..60	115 steps	Aux 1 gate range (dB)
/aux/1/gate/att	F	0..120	121 steps	Aux 1 gate attack (ms)
/aux/1/gate/hld	F	0..200	200 steps	Aux 1 gate hold (ms)
/aux/1/gate/rel	F	4..4000	130 steps	Aux 1 gate release(ms)

[31] This is for the STD eq model. Can use more parameters depending on model type

[32] Side chain EQ is not available when auxcombo [CMB] is used in dyn slot

[33] This is for the GATE gate model. Can use more parameters depending on model type

/aux/1/gate/acc	F	0..100	21 steps	Aux 1 gate accent (5)
/aux/1/gate/ratio	S		1:1.5, 1:2, 1:3, 1:4, gate	Aux 1 gate ratio
/aux/1/dynsc				Aux 1 dyn sidechain node
/aux/1/dynsc/type	S		OFF, LP12, HP12, BP, NOTCH	Aux 1 dyn sidechain type
/aux/1/dynsc/f	F	20..20000	901 steps	Aux 1 dyn sidechain frequency [Hz]
/aux/1/dynsc/q	F	0.44..10.0	181 steps	Aux 1 dyn sidechain quality
/aux/1/dynsc/src	S		SELF, BUS.1..BUS.16, MAIN.1..MAIN.4, MTX.1.. MTX.8, AUX.1.. AUX.8	Aux 1 dyn sidechain source
/aux/1/dynsc/tap	S		BUS, DYN, PFL, AFL, EQ, INS2	Aux 1 dyn sidechain tap point
/aux/1/dynsc/$solo	I	0..1		Aux 1 dyn sidechain solo
/aux/1/preins	N			Aux 1 pre-insert node
/aux/1/preins/on	I	0..1		Aux 1 pre-insert switch
/aux/1/preins/ins	S		NONE, FX1..FX16	Aux 1 pre-insert FX slot
/aux/1/preins/$stat	S		-, OK, N/A	Aux 1 pre-insert status [RO]
/aux/1/main	N			Aux 1 Main node
/aux/1/main/1	N	1..4		Aux 1 Main 1 node
/aux/1/main/1/on	I	0..1		Aux 1 Main 1 on switch
/aux/1/main/1/lvl	F	-144..10	-oo..10 in 1024 steps	Aux 1 Main 1 fader level (dB)
/aux/1/main/1/pre	I	0..1		Aux 1 sent pre fader to Main 1
/aux/1/send	N			Aux 1 sends node
/aux/1/send/1	N	1..16		Aux 1 sends 1 node
/aux/1/send/1/on	I	0..1		Aux 1 sends 1 on switch
/aux/1/send/1/lvl	F	-144..10	-oo..10 in 1024 steps	Aux 1 sends 1 fader level (dB)
/aux/1/send/1/pon	I	0..1		Aux 1 sends 1 pre always on switch
/aux/1/send/1/mode	S		PRE, POST, GRP	Aux 1 sends 1 mode
/aux/1/send/1/plink	I	0..2		Aux 1 sends 1 pan link (0=individual)
/aux/1/send/1/pan	F	-100..100	201 steps	Aux 1 sends 1 pan
/aux/1/send/MX<x>	N	<x>:1..8		Aux 1 sends matrix <x> node
/aux/1/send/MX<x>/on	I	0..1		Aux 1 sends mtx on switch
/aux/1/send/MX<x>/lvl	F	-144..10	-oo..10 in 1024 steps	Aux 1 sends mtx fader level (dB)
/aux/1/send/MX<x>/pon	I	0..1		Aux 1 sends mtx pre always on switch (ignore channel mute)
/aux/1/send/MX<x>/mode	S		PRE, POST, GRP	Aux 1 sends mtx mode
/aux/1/send/MX<x>/plink	I	0..2		Aux 1 sends mtx pan link (0=individual)
/aux/1/send/MX<x>/pan[34]	F	-100..100	201 steps	Aux 1 sends mtx pan
/aux/1/tags	S	80 chars max		Aux 1 tags[35]
/aux/1/$fdr	F	-144..10	-oo..10 in 1024 steps	Aux 1 fader level as affected by dca (dB)[RO]
/aux/1/$mute	I	0..2		Aux 1 mute {RO]
/aux/1/$muteovr	I	0..1		Aux 1 mute override

[34] plink must be 0 for this parameter to be settable

[35] Tags #D1..#D16 are 'reserved' for DCA1..16 assignment

Bus Settings

Command	Type	Range	Text	Description
/bus	N			Bus node
/bus/1	N	1..16		Bus 1 node
/bus/1/in	N			Bus 1 input node
/bus/1/in/set	N			Bus 1 input set node
/bus/1/in/set/inv	I	0..1		Bus 1 input phase invert
/bus/1/in/set/trim	F	-18..18	361 steps	Bus 1 input trim (dB)
/bus/1/in/set/bal	F	-9..9	181 steps	Bus 1 input balance (dB)
/bus/1/col	I	1..18		Bus 1 color
/bus/1/name	S		16 chars max	Bus 1 name
/bus/1/icon	I	0..999		Bus 1 icon
/bus/1/led	I	0..1		Bus 1 scribble light
/bus/1/$col	I	1..18		Bus 1 color [RO] reflects linked source or current strip value
/bus/1/$name	S		16 chars max	Bus 1 name [RO] reflects linked source or current strip value
/bus/1/$icon	I	0..999		Bus 1 icon [RO] reflects linked source or current strip value
/bus/1/busmono	I	0..1		Bus 1 mono switch
/bus/1/mute	I	0..1		Bus 1 mute
/bus/1/fdr	F	-144..10	-oo..10 in 1024 steps	Bus 1 fader level (dB)
/bus/1/pan	F	-100..100	201 steps	Bus 1 pan
/bus/1/wid	F	-150..150	61 steps	Bus 1 width (%)
/bus/1/$solo	I	0..1		Bus 1 solo
/bus/1/$sololed	I	0..2		Bus 1 solo LED {RO]
/bus/1/mon	S		A, B, A+B	Bus 1 monitor mode
/bus/1/eq	N			Bus 1 EQ node
/bus/1/eq/on	I	0..1		Bus 1 EQ on switch
/bus/1/eq/mdl	S		STD, SOUL, E88, E84, F110, PULSAR, PIA	Bus 1 EQ model (see Appendix on EQ plugins for parameters details, OSC patterns in italic below correspond to STD)
/bus/1/eq/mix	F	0..100	126 steps	Bus 1 EQ mix
/bus/1/eq/$solo	I	0..1		Bus 1 EQ solo
/bus/1/eq/$solobd	I	0..1		Bus 1 EQ band solo
/bus/1/eq/lg[36]	F	-15..15	301 steps	Bus 1 EQ low gain (dB)
/bus/1/eq/lf	F	20..2000	641 steps	Bus 1 EQ low frequency (Hz)
/bus/1/eq/lq	F	0.44..10	181 steps	Bus 1 EQ low Q
/bus/1/eq/leq	S		PEQ, SHV, CUT, BW6, BW12, BS12, LR12, BW18, BW24, BS24, LR24, BW48, LR48	Bus 1 EQ low type
/bus/1/eq/1g	F	-15..15	301 steps	Bus 1 EQ band 1 gain (dB)
/bus/1/eq/1f	F	20..20000	961 steps	Bus 1 EQ band 1 frequency (Hz)
/bus/1/eq/1q	F	0.44..10	181 steps	Bus 1 EQ band 1 Q
/bus/1/eq/2g	F	-15..15	301 steps	Bus 1 EQ band 2 gain (dB)
/bus/1/eq/2f	F	20..20000	961 steps	Bus 1 EQ band 2 frequency (Hz)
/bus/1/eq/2q	F	0.44..10	181 steps	Bus 1 EQ band 2 Q
/bus/1/eq/3g	F	-15..15	301 steps	Bus 1 EQ band 3 gain (dB)

[36] This is for the STD eq model. Can use more parameters depending on model type

/bus/1/eq/3f	F	20..20000	961 steps	Bus 1 EQ band 3 frequency (Hz)
/bus/1/eq/3q	F	0.44..10	181 steps	Bus 1 EQ band 3 Q
/bus/1/eq/4g	F	-15..15	301 steps	Bus 1 EQ band 4 gain (dB)
/bus/1/eq/4f	F	20..20000	961 steps	Bus 1 EQ band 4 frequency (Hz)
/bus/1/eq/4q	F	0.44..10	181 steps	Bus 1 EQ band 4 Q
/bus/1/eq/5g	F	-15..15	301 steps	Bus 1 EQ band 5 gain (dB)
/bus/1/eq/5f	F	20..20000	961 steps	Bus 1 EQ band 5 frequency (Hz)
/bus/1/eq/5q	F	0.44..10	181 steps	Bus 1 EQ band 5 Q
/bus/1/eq/6g	F	-15..15	301 steps	Bus 1 EQ band 6 gain (dB)
/bus/1/eq/6f	F	20..20000	961 steps	Bus 1 EQ band 6 frequency (Hz)
/bus/1/eq/6q	F	0.44..10	181 steps	Bus 1 EQ band 6 Q
/bus/1/eq/hg	F	-15..15	301 steps	Bus 1 EQ high gain (dB)
/bus/1/eq/hf	F	50..20000	833 steps	Bus 1 EQ high frequency (Hz)
/bus/1/eq/hq	F	0.44..10	181 steps	Bus 1 EQ high Q
/bus/1/eq/heq	S		PEQ, SHV, CUT, BW6, BW12, BS12, LR12, BW18, BW24, BS24, LR24, BW48, LR48	Bus 1 EQ high type
/bus/1/eq/tilt	F	-6..6	49 steps	Bus 1 EQ tilt level
/bus/1/dyn	N			Bus 1 dynamic (compressor) node
/bus/1/dyn/on	I	0..1		Bus 1 compressor switch
/bus/1/dyn/mdl	S		GATE, DUCK, E88, 9000G, D241G, DS902, DEQ, DEQ2, WAVE, PSE, CMB, RIDE, WARM, COMP, EXP, B160, B560, D241C, ECL33, 9000C, SBUS, RED3, 76LA, LA, F670, BLISS, NSTR, 2250, L100, E88C, LMT, ONEC [37]	Bus 1 compressor model, (see Appendix on Compressor plugins for parameters details, OSC patterns in italic below correspond to COMP)
/bus/1/dyn/mix	F	0..100	101 steps	Bus 1 compressor mix (%)
/bus/1/dyn/gain	F	-6..12	37 steps	Bus 1 compressor gain (dB)
/bus/1/dyn/thr[38]	F	-60..0	121 steps	Bus 1 compressor threshold (dB)
/bus/1/dyn/ratio	F	1.1..100		Bus 1 compressor ratio
/bus/1/dyn/knee	I	0..5		Bus 1 compressor knee
/bus/1/dyn/det	S		PEAK, RMS	Bus 1 compressor detect
/bus/1/dyn/att	F	0..120	121 steps	Bus 1 compressor attack (ms)
/bus/1/dyn/hld	F	1..200	200 steps	Bus 1 compressor hold (ms)
/bus/1/dyn/rel	F	4..4000	130 steps	Bus 1 compressor release (ms)
/bus/1/dyn/env	S		LIN, LOG	Bus 1 compressor envelope
/bus/1/dyn/auto	I	0..1		Bus 1 compressor auto switch
/bus/1/dynxo	N			Bus 1 compressor crossover node
/bus/1/dynxo/depth	F	0..20	41 steps	Bus 1 compressor crossover depth (dB)
/bus/1/dynxo/type	S		OFF, LO, HI, S6, S12, PEAK	Bus 1 compressor crossover type
/bus/1/dynxo/f	F	20..20000	901 steps	Bus 1 compressor crossover frequency (Hz)
/bus/1/dynxo/q	F	0.40..4.0	121 steps	Bus 1 compressor crossover quality
/bus/1/dynxo/$solo	I	0..1		Bus 1 compressor crossover solo

[37] Side chain EQ is not available when auxcombo [CMB] is used in dyn slot

[38] This is for the COMP dyn model. Can use more parameters depending on model type

/bus/1/dynsc	N			Bus 1 compressor sidechain node
/bus/1/dynsc/type	S		OFF, LP12, HP12, BP, NOTCH	Bus 1 compressor sidechain type
/bus/1/dynsc/f	F	20..20000	901 steps	Bus 1 compressor sidechain frequency (Hz)
/bus/1/dynsc/q	F	0.44..10	181 steps	Bus 1 compressor sidechain Q
/bus/1/dynsc/src	S		SELF, BUS.1..BUS.16, MAIN.1..MAIN.4, MTX.1..MTX.8, AUX.1..AUX.8	Bus 1 compressor sidechain source
/bus/1/dynsc/tap	S		BUS, DYN, PFL, AFL, EQ, INS2	Bus 1 compressor sidechain tap
/bus/1/dynsc/$solo	I	0..1		Bus 1 compressor sidechain solo
/bus/1/preins	N			Bus 1 pre-insert node
/bus/1/preins/on	I	0..1		Bus 1 pre-insert switch
/bus/1/preins/ins	S		NONE, FX1..FX16	Bus 1 pre-insert FX slot
/bus/1/preins/$stat	S		-, OK, N/A	Bus 1 pre-insert status [RO]
/bus/1/main	N			Bus 1 Main node
/bus/1/main/1	N	1..4		Bus 1 Main 1 node
/bus/1/main/1/on	I	0..1		Bus 1 Main 1 on switch
/bus/1/main/1/lvl	F	-144..10	-oo..10 in 1024 steps	Bus 1 Main 1 fader level (dB)
/bus/1/main/1/pre	I	0..1		Bus 1 sent pre fader to Main 1
/bus/1/send	N			Bus 1 sends node
/bus/1/send/1	N	1..16		Bus 1 sends 1 node
/bus/1/send/1/on	I	0..1		Bus 1 sends 1 on switch[39]
/bus/1/send/1/lvl	F	-144..10	-oo..10 in 1024 steps	Bus 1 sends 1 fader level (dB)
/bus/1/send/1/pre	I	0..1		Bus 1 sends 1 pre/post switch
/bus/1/send/MX<x>	N	<x>:1..8		Bus 1 matrix <x> sends node
/bus/1/send/MX<x>/on	I	0..1		Bus 1 mtx on switch
/bus/1/send/MX<x.>/lvl	F	-144..10	-oo..10 in 1024 steps	Bus 1 mtx fader level (dB)
/bus/1/send/MX<x>/pre	I	0..1		Bus 1 mtx pre/post switch
/bus/1/postins	N			Bus 1 post insert node
/bus/1/postins/on	I	0..1		Bus 1 post insert on switch
/bus/1/postins/ins	S		NONE, FX1..FX16	Bus 1 post insert mode
/bus/1/postins/$stat	S		-, OK, N/A	Bus 1 post insert status [RO]
/bus/1/dly	N			Bus 1 delay node
/bus/1/dly/on	I	0..1		Bus 1 delay on
/bus/1/dly/mode	S		M, FT, MS, SMP	Meters, feet, milliseconds, samples
/bus/1/dly/dly	F	0..150 m / 0.5..500 ft/ 0.5..500 ms/ 16..500 smp	1501 steps / 1000 steps / 4996 steps / 485 steps	Bus 1 delay (meters, feet, ms, samples)
/bus/1/tags	S		80 chars max	Bus 1 tags[40]
/bus/1/$fdr	F	-144..10	-oo..10 in 1024 steps	Bus 1 fader level as affected by dca (dB)[RO]

[39] Sending bus 'n' to bus 'n' is not possible. For ex. trying to enable `/bus/1/send/1/on` will be ignored.

[40] Tags #D1..#D16 are 'reserved' for DCA1..16 assignment

/bus/1/$mute	I	0..2		Bus 1 mute [RO]
/bus/1/$muteovr	I	0..1		Bus 1 mute override

Mains Settings

Command	Type	Range	Text	Description
/main	N			Main node
/main/1	N	1..4		Main 1 node
/main/1/in	N			Main 1 input node
/main/1/in/set	N			Main 1 input set node
/main/1/in/set/inv	I	0..1		Main 1 input phase invert switch
/main/1/in/set/trim	F	-18..18	361 steps	Main 1 input trim
/main/1/in/set/bal	F	-9..9	181 steps	Main 1 input balance
/main/1/col	I	1..18		Main 1 color
/main/1/name	S		16 chars max	Main 1 name
/main/1/icon	I	0..999		Main 1 icon
/main/1/led	I	0..1		Main 1 scribble light
/main/1/$col	I	1..18		Main 1 color [RO] reflects linked source or current strip value
/main/1/$name	S		16 chars max	Main 1 name [RO] reflects linked source or current strip value
/main/1/$icon	I	0..999		Main 1 icon [RO] reflects linked source or current strip value
/main/1/busmono	I	0..1		Main 1 mono switch
/main/1/mute	I	0..1		Main 1 mute
/main/1/fdr	F	-144..10	-oo..10 in 1024 steps	Main 1 fader level (dB)
/main/1/pan	F	-100..100	201 steps	Main 1 pan
/main/1/wid	F	-150..150	61 steps	Main 1 width (%)
/main/1/$solo	I	0..1		Main 1 solo switch
/main/1/$sololed	I	0..2		Main 1 solo LED [RO]
/main/1/mon	S		A, B, A+B	Main 1 monitor mode
/main/1/eq	N			Main 1 EQ node
/main/1/eq/on	I	0..1		Main 1 EQ on switch
/main/1/eq/mdl	S		STD, SOUL, E88, E84, F110, PULSAR, PIA	Main 1 EQ model, (see Appendix on EQ plugins for parameters details, OSC patterns in italic below correspond to STD)
/main/1/eq/mix	F	0..100	126 steps	Main 1 EQ mix
/main/1/eq/$solo	I	0..1		Main 1 EQ solo
/main/1/eq/$solobd	I	0..1		Main 1 EQ band solo
/main/1/eq/lg[41]	F	-15..15	301 steps	Main 1 EQ low gain (dB)
/main/1/eq/lf	F	20..2000	641 steps	Main 1 EQ low frequency (Hz)
/main/1/eq/lq	F	0.44..10	181 steps	Main 1 EQ low Q
/main/1/eq/leq	S		PEQ, SHV, CUT, BW6, BW12, BS12, LR12, BW18, BW24, BS24, LR24, BW48, LR48	Main 1 EQ low type
/main/1/eq/1g	F	-15..15	301 steps	Main 1 EQ band 1 gain (dB)
/main/1/eq/1f	F	20..20000	961 steps	Main 1 EQ band 1 frequency (Hz)
/main/1/eq/1q	F	0.44..10	181 steps	Main 1 EQ band 1 Q
/main/1/eq/2g	F	-15..15	301 steps	Main 1 EQ band 2 gain (dB)
/main/1/eq/2f	F	20..20000	961 steps	Main 1 EQ band 2 frequency (Hz)
/main/1/eq/2q	F	0.44..10	181 steps	Main 1 EQ band 2 Q
/main/1/eq/3g	F	-15..15	301 steps	Main 1 EQ band 3 gain (dB)

[41] This is for the STD eq model. Can use more parameters depending on model type

/main/1/eq/3f	F	20..20000	961 steps	Main 1 EQ band 3 frequency (Hz)
/main/1/eq/3q	F	0.44..10	181 steps	Main 1 EQ band 3 Q
/main/1/eq/4g	F	-15..15	301 steps	Main 1 EQ band 4 gain (dB)
/main/1/eq/4f	F	20..20000	961 steps	Main 1 EQ band 4 frequency (Hz)
/main/1/eq/4q	F	0.44..10	181 steps	Main 1 EQ band 4 Q
/main/1/eq/5g	F	-15..15	301 steps	Main 1 EQ band 5 gain (dB)
/main/1/eq/5f	F	20..20000	961 steps	Main 1 EQ band 5 frequency (Hz)
/main/1/eq/5q	F	0.44..10	181 steps	Main 1 EQ band 5 Q
/main/1/eq/6g	F	-15..15	301 steps	Main 1 EQ band 6 gain (dB)
/main/1/eq/6f	F	20..20000	961 steps	Main 1 EQ band 6 frequency (Hz)
/main/1/eq/6q	F	0.44..10	181 steps	Main 1 EQ band 6 Q
/main/1/eq/hg	F	-15..15	301 steps	Main 1 EQ high gain (dB)
/main/1/eq/hf	F	50..20000	833 steps	Main 1 EQ high frequency (Hz)
/main/1/eq/hq	F	0.44..10	181 steps	Main 1 EQ high Q
/main/1/eq/heq	S		PEQ, SHV, CUT, BW6, BW12, BS12, LR12, BW18, BW24, BS24, LR24, BW48, LR48	Main 1 EQ high type
/main/1/eq/tilt	F	-6..6	49 steps	Main 1 EQ tilt level
/main/1/dyn	N			Main 1 dynamic (compressor) node
/main/1/dyn/on	I	0..1		Main 1 compressor switch
/main/1/dyn/mdl	S		GATE, DUCK, E88, 9000G, D241G, DS902, DEQ, DEQ2, WAVE, PSE, CMB, RIDE, WARM, COMP, EXP, B160, B560, D241C, ECL33, 9000C, SBUS, RED3, 76LA, LA, F670, BLISS, NSTR, 2250, L100, E88C, LMT, ONEC [42]	Main 1 compressor switch, (see Appendix on Compressor plugins for parameters details, OSC patterns in italic below correspond to COMP)
/main/1/dyn/mix	F	0..100	101 steps	Main 1 compressor mix (%)
/main/1/dyn/gain	F	-6..12	37 steps	Main 1 compressor gain (dB)
/main/1/dyn/thr[43]	F	-60..0	121 steps	Main 1 compressor threshold (dB)
/main/1/dyn/ratio	F	1.1..100		Main 1 compressor ratio
/main/1/dyn/knee	I	0..5		Main 1 compressor knee
/main/1/dyn/det	S		PEAK, RMS	Main 1 compressor detect
/main/1/dyn/att	F	0..120	121 steps	Main 1 compressor attack (ms)
/main/1/dyn/hld	F	1..200	200 steps	Main 1 compressor hold (ms)
/main/1/dyn/rel	F	4..4000	130 steps	Main 1 compressor release (ms)
/main/1/dyn/env	S		LIN, LOG	Main 1 compressor envelope
/main/1/dyn/auto	I	0..1		Main 1 compressor auto switch
/main/1/dynxo	N			Main 1 compressor crossover node
/main/1/dynxo/depth	F	0..20	41 steps	Main 1 compressor crossover depth (dB)
/main/1/dynxo/type	S		OFF, LO, HI, S6, S12, PEAK	Main 1 compressor crossover type
/main/1/dynxo/f	F	20..20000	901 steps	Main 1 compressor crossover frequency (Hz)
/main/1/dynxo/q	F	0.40..4.0	121 steps	Main 1 compressor crossover quality
/main/1/dynxo/$solo	I	0..1		Main 1 compressor crossover solo

[42] Side chain EQ is not available when auxcombo [CMB] is used in dyn slot

[43] This is for the COMP dyn model. Can use more parameters depending on model type

/main/1/dynsc	N			Main 1 compressor sidechain node
/main/1/dynsc/type	S		OFF, LP12, HP12, BP, NOTCH	Main 1 compressor sidechain type
/main/1/dynsc/f	F	20..20000	901 steps	Main 1 compressor sidechain frequency (Hz)
/main/1/dynsc/q	F	0.44..10	181 steps	Main 1 compressor sidechain Q
/main/1/dynsc/src	S		SELF, BUS.1..BUS.16, MAIN.1..MAIN.4, MTX.1..MTX.8, AUX.1..AUX.8	Main 1 compressor sidechain source
/main/1/dynsc/tap	S		BUS, DYN, PFL, AFL, EQ, INS2	Main 1 compressor sidechain tap
/main/1/dynsc/$solo	I	0..1		Main 1 compressor sidechain solo
/main/1/preins	N			Main 1 pre-insert node
/main/1/preins/on	I	0..1		Main 1 pre-insert switch
/main/1/preins/ins	S		NONE, FX1..FX16	Main 1 pre-insert FX slot
/main/1/preins/$stat	S		-, OK, N/A	Main 1 pre-insert status [RO]
/main/1/send	N			Main 1 sends node
/main/1/send/MX<x>	N	<x>: 1..8		Main 1 matrix <x> sends node
/main/1/send/MX<x>/on	I	0..1		Main 1 mtx on switch
/main/1/send/MX<x>/lvl	F	-144..10	-oo..10 in 1024 steps	Main 1 mtx fader level (dB)
/main/1/send/MX<x>/pre	I	0..1		Main 1 mtx pre/post switch
/main/1/postins	N			Main 1 post insert node
/main/1/postins/on	I	0..1		Main 1 post insert on switch
/main/1/postins/ins	S		NONE, FX1..FX16	Main 1 post insert mode
/main/1/postins/$stat	S		-, OK, N/A	Main 1 post insert status [RO]
/main/1/dly	N			Main 1 delay node
/main/1/dly/on	I	0..1		Main 1 delay on switch
/main/1/dly/mode	S		M, FT, MS, SMP	Meters, feet, milliseconds, samples
/main/1/dly/dly	F	0..150 m / 0.5..500 ft/ 0.5..500 ms/ 16..500 smp	1501 steps / 1000 steps / 4996 steps / 485 steps	Main 1 delay (meters, feet, ms, samples)
/main/1/tags				
/main/1/$fdr	S		80 chars max	Main 1 tags[44]
/main/1/$mute	F	-144..10	-oo..10 in 1024 steps	Main 1 fader level as affected by dca (dB)[RO]
/main/1/$muteovr	I	0..2		Main 1 mute [RO]
	I	0..1		Main 1 mute override

[44] Tags #D1..#D16 are 'reserved' for DCA1..16 assignment

Matrix Settings

Command	Type	Range	Text	Description
/mtx	N			Matrix node
/mtx/1	N	1..8		Matrix 1 node
/mtx/1/in	N			Matrix 1 input node
/mtx/1/in/set	N			Matrix 1 input set node
/mtx/1/in/set/inv	I	0..1		Matrix 1 input phase invert
/mtx/1/in/set/trim	F	-18..18	361 steps	Matrix 1 input trim
/mtx/1/in/set/bal	F	-9..9	181 steps	Matrix 1 input balance
/mtx/1/dir	N			Matrix 1 direct input signal
/mtx/1/dir/on	I	0..1		Matrix 1 direct in on switch
/mtx/1/dir/lvl	F	-144..10	-oo..10 in 1024 steps	Matrix 1 direct in fader level (dB)
/mtx/1/dir/inv	I	0..1		Matrix 1 direct in invert
/mtx/1/dir/in	S		OFF, AES, MON.PH, MON.SPK, MON.BUS	Matrix 1 direct in input source
/mtx/1/col	I	1..18		Matrix 1 color
/mtx/1/name	S		16 chars max	Matrix 1 name
/mtx/1/icon	I	0..999		Matrix 1 icon
/mtx/1/led	I	0..1		Matrix 1 scribble light
/mtx/1/$col	I	1..18		Matrix 1 color [RO] reflects linked source or current strip value
/mtx/1/$name	S		16 chars max	Matrix 1 name [RO] reflects linked source or current strip value
/mtx/1/$icon	I	0..999		Matrix 1 icon [RO] reflects linked source or current strip value
/mtx/1/busmono	I	0..1		Matrix 1 mono switch
/mtx/1/mute	I	0..1		Matrix 1 mute
/mtx/1/fdr	F	-144..10	-oo..10 in 1024 steps	Matrix 1 fader level (dB)
/mtx/1/pan	F	-100..100	201 steps	Matrix 1 pan
/mtx/1/wid	F	-150..150	61 steps	Matrix 1 width (%)
/mtx/1/$solo	I	0..1		Matrix 1 solo switch
/mtx/1/$sololed	I	0..2		Matrix 1 solo LED [RO]
/mtx/1/mon	S		A, B, A+B	Matrix 1 monitor mode
/mtx/1/eq	N			Matrix 1 EQ node
/mtx/1/eq/on	I	0..1		Matrix 1 EQ on switch
/mtx/1/eq/mdl	S		STD, SOUL, E88, E84, F110, PULSAR, PIA	Matrix 1 EQ model, (see Appendix on EQ plugins for parameters details, OSC patterns in italic below correspond to STD)
/mtx/1/eq/mix	F	0..125	126 steps	Matrix 1 EQ mix (%)
/mtx/1/eq/$solo	I	0..1		Matrix 1 EQ solo
/mtx/1/eq/$solobd	I	0..8		Matrix 1 EQ solo band
/mtx/1/eq/lg[45]	F	-15..15	301 steps	Matrix 1 EQ low gain (dB)
/mtx/1/eq/lf	F	20..2000	641 steps	Matrix 1 EQ low frequency
/mtx/1/eq/lq	F	0.44..10	181 steps	Matrix 1 EQ low Q
/mtx/1/eq/leq	S		PEQ, SHV, CUT, BW6, BW12, BS12, LR12, BW18, BW24, BS24, LR24, BW48, LR48	Matrix 1 EQ low type

[45] This is for the STD eq model. Can use more parameters depending on model type

/mtx/1/eq/1g	F	-15..15	301 steps	Matrix 1 EQ band 1 gain (dB)
/mtx/1/eq/1f	F	20..20000	961 steps	Matrix 1 EQ band 1 frequency (Hz)
/mtx/1/eq/1q	F	0.44..10	181 steps	Matrix 1 EQ band 1 Q
/mtx/1/eq/2g	F	-15..15	301 steps	Matrix 1 EQ band 2 gain (dB)
/mtx/1/eq/2f	F	20..20000	961 steps	Matrix 1 EQ band 2 frequency (Hz)
/mtx/1/eq/2q	F	0.44..10	181 steps	Matrix 1 EQ band 2 Q
/mtx/1/eq/3g	F	-15..15	301 steps	Matrix 1 EQ band 3 gain (dB)
/mtx/1/eq/3f	F	20..20000	961 steps	Matrix 1 EQ band 3 frequency (Hz)
/mtx/1/eq/3q	F	0.44..10	181 steps	Matrix 1 EQ band 3 Q
/mtx/1/eq/4g	F	-15..15	301 steps	Matrix 1 EQ band 4 gain (dB)
/mtx/1/eq/4f	F	20..20000	961 steps	Matrix 1 EQ band 4 frequency (Hz)
/mtx/1/eq/4q	F	0.44..10	181 steps	Matrix 1 EQ band 4 Q
/mtx/1/eq/5g	F	-15..15	301 steps	Matrix 1 EQ band 5 gain (dB)
/mtx/1/eq/5f	F	20..20000	961 steps	Matrix 1 EQ band 5 frequency (Hz)
/mtx/1/eq/5q	F	0.44..10	181 steps	Matrix 1 EQ band 5 Q
/mtx/1/eq/6g	F	-15..15	301 steps	Matrix 1 EQ band 6 gain (dB)
/mtx/1/eq/6f	F	20..20000	961 steps	Matrix 1 EQ band 6 frequency (Hz)
/mtx/1/eq/6q	F	0.44..10	181 steps	Matrix 1 EQ band 6 Q
/mtx/1/eq/hg	F	-15..15	301 steps	Matrix 1 EQ high gain (dB)
/mtx/1/eq/hf	F	50..20000	833 steps	Matrix 1 EQ high frequency (Hz)
/mtx/1/eq/hq	F	0.44..10	181 steps	Matrix 1 EQ high Q
/mtx/1/eq/heq	S		PEQ, SHV, CUT, BW6, BW12, BS12, LR12, BW18, BW24, BS24, LR24, BW48, LR48	Matrix 1 EQ high type
/mtx/1/eq/tilt	F	-6..6	49 steps	Matrix 1 EQ tilt level (dB)
/mtx/1/dyn	N			Matrix 1 dynamic (compressor) node
/mtx/1/dyn/on	I	0..1		Matrix 1 compressor switch
/mtx/1/dyn/mdl	S		GATE, DUCK, E88, 9000G, D241G, DS902, DEQ, DEQ2, WAVE, PSE, CMB, RIDE, WARM, COMP, EXP, B160, B560, D241C, ECL33, 9000C, SBUS, RED3, 76LA, LA, F670, BLISS, NSTR, 2250, L100, E88C, LMT, ONEC [46]	Matrix 1 compressor model, (see Appendix on Compressor plugins for parameters details, OSC patterns in italic below correspond to COMP)
/mtx/1/dyn/mix	F	0..100	101 steps	Matrix 1 compressor mix (%)
/mtx/1/dyn/gain	F	-6..12	37 steps	Matrix 1 compressor gain (dB)
/mtx/1/dyn/thr[47]	F	-60..0	121 steps	Matrix 1 compressor threshold (dB)
/mtx/1/dyn/ratio	F	1.1..100		Matrix 1 compressor ratio
/mtx/1/dyn/knee	I	0..5		Matrix 1 compressor knee
/mtx/1/dyn/det	S		PEAK, RMS	Matrix 1 compressor detect
/mtx/1/dyn/att	F	0..120	121 steps	Matrix 1 compressor attack (ms)
/mtx/1/dyn/hld	F	1..200	200 steps	Matrix 1 compressor hold (ms)
/mtx/1/dyn/rel	F	4..4000	130 steps	Matrix 1 compressor release (ms)
/mtx/1/dyn/env	S		LIN, LOG	Matrix 1 compressor envelope
/mtx/1/dyn/auto	I	0..1		Matrix 1 compressor auto switch
/mtx/1/dynxo	N			Matrix 1 compressor crossover node

[46] Side chain EQ is not available when auxcombo [CMB] is used in dyn slot

[47] This is for the COMP dyn model. Can use more parameters depending on model type

/mtx/1/dynxo/depth	F	0..20	41 steps	Matrix 1 compressor crossover depth (dB)
/mtx/1/dynxo/type	S		OFF, LO, HI, S6, S12, PEAK]	Matrix 1 compressor crossover type
/mtx/1/dynxo/f	F	20..20000	901 steps	Matrix 1 compressor crossover frequency (Hz)
/mtx/1/dynxo/q	F	0.40..4.0	121 steps	Matrix 1 compressor crossover quality
/mtx/1/dynxo/$solo	I	0..1		Matrix 1 compressor crossover solo
/mtx/1/dynsc	N			Matrix 1 compressor sidechain node
/mtx/1/dynsc/type	S		OFF, LP12, HP12, BP, NOTCH	Matrix 1 compressor sidechain type
/mtx/1/dynsc/f	F	20..20000	901 steps	Matrix 1 compressor sidechain frequency (Hz)
/mtx/1/dynsc/q	F	0.44..10	181 steps	Matrix 1 compressor sidechain Q
/mtx/1/dynsc/src	S		SELF, BUS.1..BUS.16, MAIN.1..MAIN.4, MTX.1..MTX.8, AUX.1..AUX.8	Matrix 1 compressor sidechain source
/mtx/1/dynsc/tap	S		BUS, DYN, PFL, AFL, EQ, INS2	Matrix 1 compressor sidechain tap
/mtx/1/dynsc/$solo	I	0..1		Matrix 1 compressor sidechain solo
/mtx/1/preins	N			Matrix 1 pre-insert node
/mtx/1/preins/on	I	0..1		Matrix 1 pre-insert switch
/mtx/1/preins/ins	S		NONE, FX1..FX16	Matrix 1 pre-insert FX slot
/mtx/1/preins/$stat	S		-,OK, N/A	Matrix 1 pre-insert status [RO]
/mtx/1/postins	N			Matrix 1 post insert node
/mtx/1/postins/on	I	0..1		Matrix 1 post insert on switch
/mtx/1/postins/ins	S		NONE, FX1..FX16	Matrix 1 post insert mode
/mtx/1/postins/$stat	S		-,OK, N/A	Matrix 1 post insert status [RO]
/mtx/1/dly	N			Matrix 1 delay node
/mtx/1/dly/on	I	0..1		Matrix 1 delay on switch
/mtx/1/dly/mode	S		M, FT, MS, SMP	Meters, feet, milliseconds, samples
/mtx/1/dly/dly	F	0..150 m / 0.5..500 ft/ 0.5..500 ms/ 16..500 smp	1501 steps / 1000 steps / 4996 steps / 485 steps	Matrix 1 delay (meters, feet, ms, samples)
/mtx/1/tags	S		80 chars max	Matrix 1 tags[48]
/mtx/1/$fdr	F	-144..10	-oo..10 in 1024 steps	Matrix 1 fader level as affected by dca (dB)[RO]
/mtx/1/$mute	I	0..2		Matrix 1 mute [RO]
/mtx/1/$muteovr	I	0..1		Matrix 1 mute override

[48] Tags #D1..#D16 are 'reserved' for DCA1..16 assignment

DCA Settings

Command	Type	Range	Text	Description
/dca	N			DCA node
/dca/1	N	1..16		DCA 1 node
/dca/1/name	S		8 chars max	DCA 1 name
/dca/1/col	I	1..18		DCA 1 color
/dca/1/icon	I	0..999		DCA 1 icon
/dca/1/led	I	0..1		DCA 1 scribble light
/dca/1/mute	I	0..1		DCA 1 mute
/dca/1/fdr	F	-144..10	-oo..10 in 1024 steps	DCA 1 fader (dB)
/dca/1/$solo	I	0..1		DCA 1 solo
/dca/1/$sololed	I	0..1		DCA 1 solo LED [RO]
/dca/1/mon	S		A, B, A+B	DCA 1 monitor mode

Mutegroup Settings

Command	Type	Range	Text	Description
/mgrp	N			Mutegroup node
/mgrp/1	N	1..8		Mutegroup 1 node
/mgrp/1/name	S		8 chars max	Mutegroup 1 name
/mgrp/1/mute	I	0..1		Mutegroup 1 mute

Effects Settings

Command	Type	Range	Text	Description
/fx	N			FX node
/fx/1	N	1..16		FX 1 node
/fx/1/mdl	S		For /fx/1../fx/8: NONE, EXT, HALL, ROOM, CHAMBER, PLATE, CONCERT, AMBI, V-ROOM, V-REV, V-PLATE, GATED, REVERSE, DEL/REV, SHIMMER, SPRING, DIMCRS, CHORUS, FLANGER, ST-DL, TAP-DL, TAPE-DL, OILCAN, BBD-DL, PITCH, D-PITCH, VSS3, BPLATE, GEQ, PIA, DOUBLE, PCORR, LIMITER, DE-S2, ENHANCE, EXCITER, P-BASS, ROTARY, PHASER, PANNER, TAPE, MOOD, SUB, RACKAMP, UKROCK, ANGEL, JAZZC, DELUXE, BODY, SOUL, E88, E84, F110, PULSAR, MACH4, C5-CMB, SUB-M, V-IMG, SPKMAN, DEQ3, *EVEN*, *SOUL*, *VINTAGE*, *BUS*, *MASTER* For /fx/9../fx/16: NONE, EXT, GEQ, PIA, DOUBLE, PCORR, LIMITER, DE-S2, ENHANCE, EXCITER, P-BASS, ROTARY, PHASER, PANNER, TAPE, MOOD, SUB, RACKAMP, UKROCK, ANGEL, JAZZC, DELUXE, BODY, SOUL, E88, E84, F110, PULSAR, MACH4, C5-CMB, SUB-M, V-IMG, SPKMAN, DEQ3, *EVEN*, *SOUL*, *VINTAGE*, *BUS*, *MASTER*	FX 1 model (see Appendix for details, graphics and parameter values)
/fx/1/fxmix	F	0..100	101 steps	FX 1 mix % (depends on FX)
/fx/1/$esrc	I	0..400		FX 1 source [RO]
/fx/1/$emode	S		M, ST, M/S	FX 1 mode [RO]
/fx/1/$a_chn	I	0..76		FX 1 channel assigned to it [RO]
/fx/1/$a_pos	I	0..1		FX 1 channel insert (0=pre, 1=post) [RO]
/fx/1/…	/fx/1/… contains up to 40[49] parameters that depend on the selected model (/fx/1/mdl), as listed in the Appendix section. For the current version of FW, the highest parameter number used is 33			

[49] 1..40 are for FX parameters, 41 is for FX Mix

Cards Settings

Command	Type	Range	Text	Description
/cards	N			Cards node
/cards/$type	S		NONE, WLIVE, WDANTE, WMADI, WDWSG	Cards type [RO]
/cards/$ver	S		32 chars max	Cards version [RO]
/cards/wlive	N			Cards W-Live node
/cards/wlive/sdlink	S		IND, PAR	Cards W-Live SD parallel mode
/cards/wlive/$actlink	S		IND, PAR	Cards W-Live ACT link [RO]
/cards/wlive/$battstate	S		NONE, GOOD, LOW	Cards W-Live battery status [RO]
/cards/wlive/autoin	S		OFF, 1, 2	Cards W-Live auto input
/cards/wlive/meters	I	0..1		Cards show meters
/cards/wlive/auto_stop	S		KEEP, MAIN, ALT	Cards W-Live settings when Stop
/cards/wlive/auto_play	S		KEEP, MAIN, ALT	Cards W-Live settings when Play
/cards/wlive/auto_rec	S		KEEP, MAIN, ALT	Cards W-Live settings when Rec
/cards/wlive/1	N	1..2		Cards W-Live 1 node
/cards/wlive/1/$ctl	N			Cards W-Live 1 ctl node
/cards/wlive/1/$ctl/control	S		STOP, PPAUSE, PLAY, REC	Cards W-Live 1 control
/cards/wlive/1/$ctl/opensession	I	0..100		Cards W-Live 1 open session #
/cards/wlive/1/$ctl/editmarker	I	0..100		Cards W-Live 1 edit marker (set marker to current **PAUSE** time or last start **PLAY** time)
/cards/wlive/1/$ctl/gotomarker	I	0..101		Cards W-Live 1 goto marker # 101 is used to validate stime data
/cards/wlive/1/$ctl/deletemarker	I	0..100		Cards W-Live 1 delete marker #
/cards/wlive/1/$ctl/deletesession	I	0..100		Cards W-Live 1 delete session #
/cards/wlive/1/$ctl/stime	F	0..36000000	36000000 steps	Cards W-Live 1 time (ms). Must be followed by a **`$ctl/gotomarker 101`** to be taken into account
/cards/wlive/1/$ctl/namesession	S		19 chars max	Cards W-Live 1 name session. Works only in **STOP** mode.
/cards/wlive/1/$ctl/setmarker	I	0..1		Cards W-Live 1 set marker
/cards/wlive/1/$ctl/formatsdcard	I	0..1		Cards W-Live 1 format SD card
/cards/wlive/1/cfg	N			Cards W-Live 1 cfg node
/cards/wlive/1/cfg/rectracks	S		32, 16, 8	Cards W-Live 1 rec tracks
/cards/wlive/1/cfg/playmode	S		PLAY, A->B, LOOP	Cards W-Live 1 play mode
/cards/wlive/1/$stat	N			Cards W-Live 1 status node
/cards/wlive/1/$stat/state	S		STOP, PPAUSE, PLAY, REC	Cards W-Live 1 state [RO]
/cards/wlive/1/$stat/etime	F	0..36000000	36000000 steps	Cards W-Live 1 etime (current time)
/cards/wlive/1/$stat/sdfree	F	0..36000000	36000000 steps	Cards W-Live 1 SD free time
/cards/wlive/1/$stat/sdsize	I	0..1024		Cards W-Live 1 SD size (Gb) [RO]
/cards/wlive/1/$stat/sdstate	S		NONE, READY, PROTECT, ERASE, ERROR	Cards W-Live 1 SD state [RO]

/cards/wlive/1/$stat/sessionlist	S		Ex: 2020-04-04 10:16:36, 2020-01-27 19:59:02, …	Cards W-Live 1 list of session recorded date and time
/cards/wlive/1/$stat/markerlist	S			Cards W-Live 1 current marker time
/cards/wlive/1/$stat/snamelist	S		Ex: CC Hard Candy Fi, CC Mr Jones	Cards W-Live 1 session names list[50] [RO]
/cards/wlive/1/$stat/sessions	I	0..100		Cards W-Live 1 total number of sessions [RO]
/cards/wlive/1/$stat/markers	I	0..100		Cards W-Live 1 total number of markers [RO]
/cards/wlive/1/$stat/sessionlen	F	0..36000000	36000000 steps	Cards W-Live 1 session length [RO]
/cards/wlive/1/$stat/sessionpos	I	0..100		Cards W-Live 1 session position
/cards/wlive/1/$stat/markerpos	I	0..100		Cards W-Live 1 marker position
/cards/wlive/1/$stat/tracks	S		32, 16, 8	Cards W-Live 1 track number in current session [RO]
/cards/wlive/1/$stat/rate	S		44.1, 48	Cards W-Live 1 sample rate [RO]
/cards/wlive/1/$stat/linkedpos	I	0..100		Cards W-Live session link position in the other card (only when a linked session is active)[51]
/cards/wlive/1/$stat/start	F	0..36000000	36000000 steps	Cards W-Live 1 start
/cards/wlive/1/$stat/stop	F	0..36000000	36000000 steps	Cards W-Live 1 stop
/cards/wlive/1/$stat/errormessage	S		32 chars max	Cards W-Live 1 error message [RO]
/cards/wlive/1/$stat/errorcode	I	0..34		Cards W-Live 1 error code [RO]
/cards/wmadi				Cards MADI node
/cards/wmadi/mode	S		SFP1, SFP2, BNC, SFP1/2, SFP1/BNC	Cards MADI operating mode
/cards/wmadi/rxclock	I	0..1		Cards MADI clock
/cards/wmadi/$sfp1stat	I	0..2		Cards MADI SFP1 status [RO]
/cards/wmadi/$sfp2stat	I	0..2		Cards MADI SFP2 status [RO]
/cards/wmadi/$bncstat	I	0..2		Cards MADI BNC status [RO]

[50] Only the first name of the list is returned by std OSC command. You must use the node definition command (OSC or native interface) to get the full contents.

[51] There can be a maximum of 100 sessions per card

USB Player Settings

Command	Type	Range	Text	Description
/play[52]	N			USB Player node
/play/$songs	S		List of strings	List of songs in the current playlist [RO][53]
/play/$actlist	S		256 chars max	Path to USB files [RO]
/play/$actidx	I		1..n	Current active entry in the playlist
/play/$actionidx	I		1..n	[RO]
/play/$playfile	S		256 chars max	Full path to a song to play using `/play/$action ,s PLAYFILE`
/play/$action	S		IDLE, STOP, PLAY, PAUSE, NEXT, PREV, PLAYFILE	USB Player action
/play/$actstate	S		STOP, PLAY, PAUSE, ERROR	USB Player active state [RO]
/play/$actfile	S		256 chars max	USB Player active file [RO]
/play/$song	S		64 chars max	USB Player song [RO]
/play/$album	S		64 chars max	USB Player album [RO]
/play/$artist	S		64 chars max	USB Player artist [RO]
/play/$pos	F	0..35999	36000 steps	USB Player position
/play/$total	F	0..35999	36000 steps	USB Player total time [RO]
/play/$resolution	S		16, 24	USB Player resolution [RO]
/play/$channels	S		1, 2, 3, 4	USB Player channels [RO]
/play/$rate	S		44.1, 48	USB Player sample rate [RO]
/play/$format	S		WAV, MP3, FLAC	USB Player format [RO]
/play/repeat	I	0..1		USB Player repeat
/rec	N			USB Recorder node
/rec/$actstate	S		STOP, REC, PAUSE, ERROR	USB Recorder active state
/rec/$actfile	S			USB Recorder active filename
/rec/$action	S		STOP, REC, PAUSE, NEWFILE	USB Recorder action
/rec/path	S			USB Recorder filename path
/rec/resolution	S		16, 24	USB Recorder resolution
/rec/channels	S		2, 4	USB Recorder channels
/rec/$time	F			USB Recorder time

[52] These commands are valid only when a playlist is active, and opened.

[53] Will provide only the first element of the list. Use the node level request to get the full list of songs in the playlist, for ex: `/play~~~,s~~?~~~`

WING ce_data OSC commands list

Control Settings, listed below, form a large set of OSC commands and parameters, all[54] under the ***ce_data*** section in `JSON` snapshot files.

Control Settings

Command	Type	Range	Text	Description
/$ctl	N			Control node
/$ctl/$stat	N			Control status node
/$ctl/$stat/selidx	I	1..76		Channel strip selected ID[55]
/$ctl/$stat/pageidx	I	0..30		Channel page ID
/$ctl/$stat/bandidx	I	1..8		Channel EQ band ID

/$ctl/$stat/sof	S	-1..76		Sends on fader (SoF) status
				`-1: sof active`
				`0: sof not active (light off)`
				`1..76: sof active on channel #`
				If/when using `int` data, values above are `+1`, so in the range `[0..77]`

/$ctl/$stat/cnslock	S		19 chars when locked 0 chars if unlocked	Console lock [RO] – The console lock string is made of 19 characters `0` or `1` depending on which screen buttons were pressed to lock the console. Characters 1 to 7 map to the buttons on the screen left, starting with `HOME` [`ASSIGN` is char #7], and characters 18 & 19 map to the buttons on the right side of the screen. Other characters are always `0`. Ex: `1001001000000000000` - buttons `HOME`, `ROUTING` and `ASSIGN` have been used to lock the desk.
/$ctl/$stat/sendpage	S		BUS, MATRIX	Bus or matrix Sends being displayed on screen
/$ctl/$stat/chsetuptab	I	1..4	1: OVERVIEW 2: ICON/COLOR 3: NAME 4: TAGS	HOME page tab being displayed on screen
/$ctl/cfg	N			Control config node
/$ctl/cfg/lights	N			Lights node
/$ctl/cfg/lights/btns	I	0..100		Buttons backlight intensity
/$ctl/cfg/lights/leds	I	5..100		Buttons/LED light intensity
/$ctl/cfg/lights/meters	I	0..100		Meters intensity
/$ctl/cfg/lights/rgbleds	I	0..100		Color LED intensity (scribble lights)
/$ctl/cfg/lights/chlcds	I	5..100		Channel LCD intensity (scribble backlight)
/$ctl/cfg/lights/chlcdctr	I	0..100		Channel LCD contrast (scribble contrast)
/$ctl/cfg/lights/chedit	I	5..100		Channel strip intensity

[54] Except for /$ctl/$stat, and parameters /$ctl/cfg/$noautosave and /$ctl/cfg/savenow

[55] The get command reports values between 0 and 75, but index 1 to 76 should be used when setting values.

/$ctl/cfg/lights/main	I	5..100		Touchscreen intensity
/$ctl/cfg/lights/glow	I	0..100		Under console light intensity
/$ctl/cfg/lights/patch	I	0..100		Patch panel light intensity
/$ctl/cfg/lights/lamp	I	0..100		Lamp light intensity
/$ctl/cfg/rta[56]	N			RTA node (view options)
/$ctl/cfg/rta/homedisp	S		OFF, 1/3, FULL	RTA home size/mode
/$ctl/cfg/rta/homecol	S		RD25, RD50, RD75, AM25, AM50, AM75, BL25, BL50, BL75	RTA home color
/$ctl/cfg/rta/hometap	S		IN, EQ, POST	RTA home tap
/$ctl/cfg/rta/eqdisp	S		Off, 1/4, 1/3, 1/2, OVL/3, OVL	RTA EQ size/mode
/$ctl/cfg/rta/eqcol	S		RD25, RD50, RD75, AM25, AM50, AM75, BL25, BL50, BL75	RTA EQ color
/$ctl/cfg/rta/cheqtap	S		PRE, POST	RTA EQ tap
/$ ctl/cfg /rta/chflttap	S		PRE, POST	RTA Channel filter tap
/$ ctl/cfg /muteovr	I	0..1		Chan strip mute overrides mute group
/$ctl/cfg/soloexcl	I	0..1		Solo exclusive
/$ctl/cfg/selfsolo	I	0..1		Select follows solo
/$ctl/cfg/solofsel	I	0..1		Solo follows select
/$ctl/cfg/sof2solo	I	0..1		Bus SOF activates solo
/$ctl/cfg/sfdr2solo	I	0..1		Send page activates solo[57]
/$ctl/cfg/layerlinkl	I	0..1		User Layer link left/center
/$ctl/cfg/layerlinkr	I	0..1		User Layer link center/right
/$ctl/cfg/autoview	I	0..1		Screen follows channel strip
/$ctl/cfg/csctouch	I	0..1		Channel strip touch select
/$ctl/cfg/autosel_L	I	0..1		Channel auto select left
/$ctl/cfg/autosel_C	I	0..1		Channel auto select center
/$ctl/cfg/autosel_R	I	0..1		Channel auto select right
/$ctl/cgf/autosel_CMPCT	I	0..1		Compact Layer
/$ctl/cgf/autosel_RCK	I	0..1		Rack Layer
/$ctl/cgf/autosel_EXT	I	0..1		Ext Layer[58]
/$ctl/cgf/autosel_VRT	I	0..1		Virtual Layer
/$ctl/cgf/fdrbanking	I	0..1		Full Fader Paging
/$ctl/cgf/fdrscrnlink	I	0..1		Fader/Screen Layer link
/$ctl/cfg/soffdr	S		L/C, ALL	SOF Faders (L/C: left/center)
/$ctl/cfg/sofbutton	S		AUTO, ON, FLASH	SOF button mode
/$ctl/cfg/sofframe	I	0..1		SOF frame
/$ctl/cfg/sofmode	I	0..1		Alternative SOF mode
/$ctl/cfg/seldblclick	S		OFF, HOME, BUSFX	Where Double click select takes you
/$ctl/cfg/usrmode	S		BUS, CC	Use F1-F3 as BUS or Custom Control
/$ctl/cfg/mfdr	S		OFF, MAIN.1, .., MAIN.4, MTX.1,.., MTX.8, DCA.1, .., DCA.16, CH.1, .. CH.40, AUX.1, .. AUX.8, BUS 1..BUS 16	What functionality is assigned to the Compact model fader 13/Main

[56] See also /cfg/rta commands

[57] Only applicable from the SENDS page (VIEW->SENDS, top of the screen)

[58] EXT: Reserved in the present FW release

/$ctl/cfg/mfdrkeep	I	0..1		Compact model main fader keep
/$ctl/cfg/cscmode	S		BUS, DCA, MAIN, USER	Operation mode for the 16 buttons of Compact model
/$ctl/cfg/rackmode	S		CH, MGRP, CC, USB, SD-A, SD-B	Operation mode for the 4 CC section on Rack model
/$ctl/cfg/busspill	I	0..1		Compact model only: 1: **push**->bus spill, **hold**->bus send 0: **push**->bus send, **hold**->bus spill
/$ctl/cfg/mainspill	I	0..1		Compact model only: 1: **push**->main spill, **hold**->main send 0: **push**->main send, **hold**->main spill
/$ctl/cfg/mtxspill	I	0..1		Compact model only: 1: **push**->mtx spill, **hold**->mtx send 0: **push**->mtx send, **hold**->mtx spill
/$ctl/cfg/dcaspill	I	0..1		1: **push**->dca spill, **hold**->dca [un]select 0: **push**->dca show, **hold**->dca [un]select
/$ctl/cfg/dcacc	I	0..1		16 User Custom Control buttons on DCA buttons activated [Compact only]
/$ctl/cfg/showfdr	I	0..1		Temporarily show fader value on respective scribble when moving faders
/$ctl/layer	N			Layer node
/$ctl/layer/L	N			Left WING [only] layer node

/$ctl/layer/L/sel	I	1..22	1..7 settable[59]
			8..9 no-op
			10..22 fixed/pre-assigned
				Left WING [only] layer select[60]
				` 1: Ch 1..Ch 12`
				` 2: Ch 13..Ch 24`
				` 3: Ch 25..Ch 36`
				` 4: Ch 36..Ch 40 / Aux 1..Aux 8`
				` 5: Bus 1..Bus 12`
				` 6: User 1`
				` 7: User 2`
				` 8: No-op`
				` 9: No-op`
				`10: Ch 1..Ch 8`
				`11: Ch 9..Ch 16`
				`12: Ch 17..Ch 24`
				`13: Ch 25..Ch 32`
				`14: Ch 33..Ch 40`
				`15: Aux 1..Aux 8`
				`16: Bus 1..Bus 8`
				`17: Bus 9..Bus 16`
				`18: Main 1..Main 4`
				`19: Matrix 1..Matrix 8`
				`20: DCA 1..DCA 8`
				`21: DCA 9..DCA 16`
				`22: spilled layer`

/$ctl/layer/L/spidx	I	0..63		Spilled group
				` 0: OFF`
				` 1..16: DCA 1..16`
				`17-32: FX 1..16`
				`33-48: BUS 1..16 [TBV]`
				`49..56: MTX 1..16 [TBV]`
				`57..60: MAIN 1..4 [TBV]`
				`61: AUTOX`
				`62: AUTOY`

[59] Full size console has 7 layers whereas Compact console has 9

[60] Full-size console layers names

				63: SOF
/$ctl/layer/L/1	N	1..7		Left WING [only] layer 1 node (see above)
/$ctl/layer/L/1/ofs	I	0..12		Left WING [only] layer 1 offset (from <4 or4> keys for ex.)
/$ctl/layer/L/1/name	S		10 chars max CH1-12, CH13-24, CH25-36, CH37-AUX, BUSES, USER1, USER2	Left WING [only] layer 1 name
/$ctl/layer/L/1/1	N	1..24		Left WING [only] layer 1, node 1
/$ctl/layer/L/1/1/type	S		OFF, CH, BUS, DCA, MIDI, SEND, FX, SENDS	Left WING [only] layer 1, node 1 type (OSC patterns in italic below correspond to MIDI type)
/$ctl/layer/L/1/1/i	I	1..127		Left WING [only] layer 1, node 1 index
/$ctl/layer/L/1/1/dst	I	1..28		Left WING [only] layer 1, node 1 destination index (used for type SEND)
/$ctl/layer/L/1/1/val	*I*	*0..127*		*Left* WING [only] *layer 1, node 1 value (when type MIDI)*
/$ctl/layer/L/$spill				Left WING [only] $spill
/$ctl/layer/L/$spill/ofs	I	0..64		Left WING [only] $spill offset
/$ctl/layer/L/$spill/name	S		10 chars max	Left WING [only] layer $spill name[61]
/$ctl/layer/L/$spill/1	N	1..76		Left WING [only] layer $spill, node 1
/$ctl/layer/L/$spill/1/type	S		OOFF, CH, BUS, DCA, MIDI, SEND, FX, SENDS	Left WING [only] layer $spill, node 1 type (OSC patterns in italic below correspond to MIDI type)
/$ctl/layer/L/$spill1/i	I	1..127		Left WING [only] layer $spill, node 1 index
/$ctl/layer/L/$spill/1/dst	I	1..28		Left WING [only] layer $spill, node 1 destination index (used for type SEND)
/$ctl/layer/L/$spill/1/val	*I*	*0..127*		*Left* WING [only] *layer $spill, node 1 value (when type MIDI)*
/$ctl/layer/C	N			Center WING [only] layer node
/$ctl/layer/C/sel	I	1..22	1..6 settable 7..9 No-op 10..22 fixed/pre-assigned	Center WING [only] layer select[62]: 1: DCA 2: Main/Matrix 3: Aux/FX 4: Bus/Master 5: User 1 6: User 2 7: No-op 8: No-op 9: No-op 10: Ch 1..Ch 8 11: Ch 9..Ch 16 12: Ch 17..Ch 24 13: Ch 25..Ch 32

[61] Can be set, but will likely be replaced by default names from console layer changes, such as AUTO-X, AUTO-Y, or SPILL

[62] Full-size console layers names

				14: Ch 33..Ch 40 15: Aux 1..Aux 8 16: Bus 1..Bus 8 17: Bus 9..Bus 16 18: Main 1..Main 4 19: Matrix 1..Matrix 8 20: DCA 1..DCA 8 21: DCA 9..DCA 16 22: spilled layer
/$ctl/layer/C/spidx	I	0..62		Spilled group 0: OFF 1..16: DCA 1..16 17-32: FX 1..16 33-48: BUS 1..16 [TBV] 49..56: MTX 1..16 [TBV] 57..60: MAIN 1..4 [TBV] 61: AUTOX 62: AUTOY 63: SOF
/$ctl/layer/C/1	N	1..9		Center WING [only] layer 1 node (see above)
/$ctl/layer/C/1/ofs	I	0..8		Center WING [only] layer 1 offset
/$ctl/layer/C/1/name	S		10 chars max DCA, MAIN, AUX, BUSES, USER1, USER2	Center WING [only] layer 1 name
/$ctl/layer/C/1/1	N	1..16		Center WING [only] layer 1, node 1
/$ctl/layer/C/1/1/type	S		OFF, CH, BUS, DCA, MIDI, SEND, FX, SENDS	Center WING [only] layer 1, node 1 type (OSC patterns in italic below correspond to MIDI type)
/$ctl/layer/C/1/1/i	I	1..127		Center WING [only] layer 1, node 1 index
/$ctl/layer/C/1/1/dst	I	1..28		Center WING [only] layer 1, node 1 destination index (used for type SEND)
/$ctl/layer/C/1/1/val	*I*	*0..127*		*Center* WING [only] *layer 1, node 1 value (when type MIDI)*
/$ctl/layer/C/$spill				Center WING [only] $spill
/$ctl/layer/C/$spill/ofs	I	0..64		Center WING [only] $spill offset
/$ctl/layer/C/$spill/name	S		10 chars max	Center WING [only] layer $spill name[63]
/$ctl/layer/C/$spill/1	N	1..76		Center WING [only] layer $spill, node 1
/$ctl/layer/C/$spill/1/type	S		OFF, CH, BUS, DCA, MIDI, SEND, FX, SENDS	Center WING [only] layer $spill, node 1 type (OSC patterns in italic below correspond to MIDI type)
/$ctl/layer/C/$spill1/i	I	1..127		Center WING [only] layer $spill, node 1 index
/$ctl/layer/C/$spill/1/dst	I	1..28		Center WING [only] layer $spill, node 1 destination index (used for type SEND)
/$ctl/layer/C/$spill/1/val	*I*	*0..127*		Center WING [only] *layer $spill, node 1 value (when type MIDI)*

[63] Can be set, but will likely be replaced by default names from console layer changes, such as AUTO-X, AUTO-Y, or SPILL

/$ctl/layer/R	N			Right WING [only] layer node
/$ctl/layer/R/sel	I	1..22	1..7 settable 8, 9 No-op 10..22 fixed/pre-assigned	Right WING [only] layer select[64]: 1: Main 2: DCA 3: Channels 4: Aux/FX 5: Bus/Master 6: User 1 7: User 2 8: No-op 9: No-op 10: Ch 1..Ch 4 11: Ch 9..Ch 12 12: Ch 17..Ch 20 13: Ch 25..Ch 28 14: Ch 33..Ch 36 15: Aux 1..Aux 4 16: Bus 1..Bus 4 17: Bus 9..Bus 12 18: Main 1..Main 4 19: Matrix 1..Matrix 4 20: DCA 1..DCA 4 21: DCA 9..DCA 12 22: spilled layer
/$ctl/layer/R/spidx	I	0..63		Spilled group 0: OFF 1..16: DCA 1..16 17-32: FX 1..16 33-48: BUS 1..16 [TBV] 49..56: MTX 1..16 [TBV] 57..60: MAIN 1..4 [TBV] 61: AUTOX 62: AUTOY 63: SOF
/$ctl/layer/R/1	N	1..9		Right WING [only] layer 1 node (see above)
/$ctl/layer/R/1/ofs	I	0..15		Right WING [only] layer 1 offset
/$ctl/layer/R/1/name	S		MAIN, DCA, CH1-40, AUX, BUSES, USER1, USER2	Right WING [only] layer 1 name
/$ctl/layer/R/1/1	N	1..16 (40 for…/R/3…)		Right WING [only] layer 1, node 1. 16 nodes except for type CH1-40: 40 nodes
/$ctl/layer/R/1/1/type	S		OFF, CH, BUS, DCA, MIDI, SEND, FX, SENDS	Right WING [only] layer 1, node 1 type (OSC patterns in italic below correspond to MIDI type)
/$ctl/layer/R/1/1/i	I	0..127		Right WING [only] layer 1, node 1 index
/$ctl/layer/R/1/1/dst	I	1..28		Right WING [only] layer 1, node 1 destination index (used for type SEND)
/$ctl/layer/R/1/1/val	*I*	*0..127*		*Right* WING [only] *layer 1, node 1 value (when type MIDI)*
/$ctl/layer/R/$spill				*Right* WING [only] $spill

[64] Full-size console layers names

/$ctl/layer/R/$spill/ofs	I	0..64		*Right* WING [only] $spill offset
/$ctl/layer/R/$spill/name	S		10 chars max	*Right* WING [only] layer $spill name[65]
/$ctl/layer/R/$spill/1	N	1..76		*Right* WING [only] layer $spill, node 1
/$ctl/layer/R/$spill/1/type	S		OFF, CH, BUS, DCA, MIDI, SEND, FX, SENDS	*Right* WING [only] layer $spill, node 1 type (OSC patterns in italic below correspond to MIDI type)
/$ctl/layer/R/$spill1/i	I	1..127		*Right* WING [only] layer $spill, node 1 index
/$ctl/layer/R/$spill/1/dst	I	1..28		*Right* WING [only] layer $spill, node 1 destination index (used for type SEND)
/$ctl/layer/R/$spill/1/val	*I*	*0..127*		*Right* WING [only] *layer $spill, node 1 value (when type MIDI)*
/$ctl/layer/CMPCT				

/$ctl/layer/CMPCT/sel	I	1..22	1..7 settable
			8..22 [TBV]
				Compact [only] layer select[66]
				`1: Ch 1..Ch 12`
				`2: Ch 13..Ch 24`
				`3: Ch 25..Ch 36`
				`4: Ch 36..Ch 40 / Aux 1..Aux 8`
				`5: Bus 1..Bus 12`
				`6: User 1`
				`7: User 2`
				`8..21: [TBV]`
				`22: spilled layer`

/$ctl/layer/CMPCT/spidx	I	0..63		Spilled group
				`0: OFF`
				`1..16: DCA 1..16`
				`17-32: FX 1..16`
				`33-48: BUS 1..16 [TBV]`
				`49..56: MTX 1..16 [TBV]`
				`57..60: MAIN 1..4 [TBV]`
				`61: AUTOX`
				`62: AUTOY`
				`63: SOF`

/$ctl/layer/CMPCT/1	N	1..9		Compact [only] layer 1 node (see above)
/$ctl/layer/CMPCT/1/ofs	I	0..12		Compact [only] layer 1 offset
/$ctl/layer/CMPCT/1/name	S		10 chars max CH1-12, CH13-24, CH25-36, CH37-AUX, BUSES, … [TBV]	Compact [only] layer 1 name
/$ctl/layer/CMPCT/1/1	N	1..24		Compact [only] layer 1, node 1
/$ctl/layer/CMPCT/1/1/type	S		OFF, CH, BUS, DCA, MIDI, SEND, FX, SENDS	Compact [only] layer 1, node 1 type (OSC patterns in italic below correspond to MIDI type)
/$ctl/layer/CMPCT/1/1/i	I	1..127		Compact [only] layer 1, node 1 index
/$ctl/layer/CMPCT/1/1/dst	I	1..28		Compact [only] layer 1, node 1 destination index (used for type SEND)
/$ctl/layer/CMPCT/1/1/val	*I*	*0..127*		Compact [only] *layer 1, node 1 value (when type MIDI)*

[65] Can be set, but will likely be replaced by default names from console layer changes, such as `AUTO-X`, `AUTO-Y`, or `SPILL`

[66] Compact console layers names

/$ctl/layer/CMPCT/$spill				Compact WING [only] $spill
/$ctl/layer/CMPCT/$spill/ofs	I	0..64		Compact WING [only] $spill offset
/$ctl/layer/CMPCT/$spill/name	S		10 chars max	Compact WING [only] layer $spill name[67]
/$ctl/layer/CMPCT/$spill/1	N	1..76		Compact WING [only] layer $spill, node 1
/$ctl/layer/CMPCT/$spill/1/type	S		OFF, CH, BUS, DCA, MIDI, SEND, FX, SENDS	Compact WING [only] layer $spill, node 1 type (OSC patterns in italic below correspond to MIDI type)
/$ctl/layer/CMPCT/$spill1/i	I	1..127		Compact WING [only] layer $spill, node 1 index
/$ctl/layer/CMPCT/$spill/1/dst	I	1..28		Compact WING [only] layer $spill, node 1 destination index (used for type SEND)
/$ctl/layer/*CMPCT*/$spill/1/val	*I*	*0..127*		Compact WING [only] *layer $spill, node 1 value (when type MIDI)*
/$ctl/layer/RCK				

/$ctl/layer/RCK/sel	I	1..22	1..7 settable
			8..22 [TBV]
				Rack [only] layer select[68]
				`1: Ch 1..Ch 12`
				`2: Ch 13..Ch 24`
				`3: Ch 25..Ch 36`
				`4: Ch 36..Ch 40 / Aux 1..Aux 8`
				`5: Bus 1..Bus 12`
				`6: User 1`
				`7: User 2`
				`8..21: [TBV]`
				`22: spilled layer`

/$ctl/layer/RCK/spidx	I	0..63		Spilled group
				`0: OFF`
				`1..16: DCA 1..16`
				`17-32: FX 1..16`
				`33-48: BUS 1..16 [TBV]`
				`49..56: MTX 1..16 [TBV]`
				`57..60: MAIN 1..4 [TBV]`
				`61: AUTOX`
				`62: AUTOY`
				`63: SOF`

/$ctl/layer/RCK/1	N	1..5		Rack [only] layer 1 node (see above)
/$ctl/layer/RCK/1/ofs	I	0..36		Rack [only] layer 1 offset
/$ctl/layer/RCK/1/name	S		10 chars max CH1-12, CH13-24, CH25-36, CH37-AUX, BUSES, … [TBV]	Rack [only] layer 1 name
/$ctl/layer/RCK/1/1	N	1..40		Rack [only] layer 1, node 1
/$ctl/layer/RCK/1/1/type	S		OOFF, CH, BUS, DCA, MIDI, SEND, FX, SENDS	Rack [only] layer 1, node 1 type (OSC patterns in italic below correspond to MIDI type)
/$ctl/layer/RCK/1/1/i	I	1..127		Rack [only] layer 1, node 1 index
/$ctl/layer/RCK/1/1/dst	I	1..28		Rack [only] layer 1, node 1 destination index (used for type SEND)

[67] Can be set, but will likely be replaced by default names from console layer changes, such as `AUTO-X`, `AUTO-Y`, or `SPILL`

[68] Rack console layers names

/$ctl/layer/RCK/1/1/val	*I*	*0..127*		Rack [only] *layer 1, node 1 value (when type MIDI)*
/$ctl/layer/RCK/$spill				Rack WING [only] $spill
/$ctl/layer/RCK/$spill/ofs	I	0..64		Rack WING [only] $spill offset
/$ctl/layer/RCK/$spill/name	S		10 chars max	Rack WING [only] layer $spill name[69]
/$ctl/layer/RCK/$spill/1	N	1..76		Rack WING [only] layer $spill, node 1
/$ctl/layer/RCK/$spill/1/type	S		OFF, CH, BUS, DCA, MIDI, SEND, FX, SENDS	Rack WING [only] layer $spill, node 1 type (OSC patterns in italic below correspond to MIDI type)
/$ctl/layer/RCK/$spill1/i	I	1..127		Rack WING [only] layer $spill, node 1 index
/$ctl/layer/RCK/$spill/1/dst	I	1..28		Rack WING [only] layer $spill, node 1 destination index (used for type SEND)
/$ctl/layer/*RCK*/$spill/1/val	*I*	*0..127*		Rack WING [only] *layer $spill, node 1 value (when type MIDI)*
/$ctl/layer/EXT				

/$ctl/layer/EXT/sel	I	1..22	1..7 settable
			8..22 [TBV]
				Extern [only] layer select[70]
				`1: Ch 1..Ch 12`
				`2: Ch 13..Ch 24`
				`3: Ch 25..Ch 36`
				`4: Ch 36..Ch 40 / Aux 1..Aux 8`
				`5: Bus 1..Bus 12`
				`6: User 1`
				`7: User 2`
				`8..21: [TBV]`
				`22: spilled layer`

/$ctl/layer/EXT/spidx	I	0..62		Spilled group
				`0: OFF`
				`1..16: DCA 1..16`
				`17-32: FX 1..16`
				`33-48: BUS 1..16 [TBV]`
				`49..56: MTX 1..16 [TBV]`
				`57..60: MAIN 1..4 [TBV]`
				`61: AUTOX`
				`62: AUTOY`

/$ctl/layer/EXT/1	N	1..8		Extern [only] layer 1 node (see above)
/$ctl/layer/EXT/1/ofs	I	0..8		Extern [only] layer 1 offset
/$ctl/layer/EXT/1/name	S		10 chars max … [TBV]	Extern [only] layer 1 name
/$ctl/layer/EXT/1/1	N	1..16		Extern [only] layer 1, node 1
/$ctl/layer/EXT/1/1/type	S		OFF, CH, BUS, DCA, MIDI, SEND, FX, SENDS	Extern [only] layer 1, node 1 type (OSC patterns in italic below correspond to MIDI type)
/$ctl/layer/EXT/1/1/i	I	1..127		Extern [only] layer 1, node 1 index
/$ctl/layer/EXT/1/1/dst	I	1..28		Extern [only] layer 1, node 1 destination index (used for type SEND)

[69] Can be set, but will likely be replaced by default names from console layer changes, such as `AUTO-X`, `AUTO-Y`, or `SPILL`

[70] Ext console layers names [TBV]

/$ctl/layer/EXT/1/1/val	*I*	*0..127*		Extern [only] *layer 1, node 1 value (when type MIDI)*
/$ctl/layer/EXT/$spill				Extern WING [only] $spill
/$ctl/layer/EXT/$spill/ofs	I	0..64		Extern WING [only] $spill offset
/$ctl/layer/EXT/$spill/name	S		10 chars max	Extern WING [only] layer $spill name[71]
/$ctl/layer/EXT/$spill/1	N	1..76		Extern WING [only] layer $spill, node 1
/$ctl/layer/EXT/$spill/1/type	S		OOFF, CH, BUS, DCA, MIDI, SEND, FX, SENDS	Extern WING [only] layer $spill, node 1 type (OSC patterns in italic below correspond to MIDI type)
/$ctl/layer/EXT/$spill1/i	I	1..127		Extern WING [only] layer $spill, node 1 index
/$ctl/layer/EXT/$spill/1/dst	I	1..28		Extern WING [only] layer $spill, node 1 destination index (used for type SEND)
/$ctl/layer/*EXT*/$spill/1/val	*I*	*0..127*		Extern WING [only] *layer $spill, node 1 value (when type MIDI)*
/$ctl/layer/VRT				

/$ctl/layer/VRT/sel	I	1..22	1..7 settable
			8..22 [TBV]
				Virtual [only] layer select[72]
				`1: Ch 1..Ch 12`
				`2: Ch 13..Ch 24`
				`3: Ch 25..Ch 36`
				`4: Ch 36..Ch 40 / Aux 1..Aux 8`
				`5: Bus 1..Bus 12`
				`6: User 1`
				`7: User 2`
				`8..21: [TBV]`
				`22: spilled layer`

/$ctl/layer/VRT/spidx	I	0..62		Spilled group
				`0: OFF`
				`1..16: DCA 1..16`
				`17-32: FX 1..16`
				`33-48: BUS 1..16 [TBV]`
				`49..56: MTX 1..16 [TBV]`
				`57..60: MAIN 1..4 [TBV]`
				`61: AUTOX`
				`62: AUTOY`

/$ctl/layer/VRT/1	N	1..8		Virtual [only] layer 1 node (see above)
/$ctl/layer/VRT/1/ofs	I	0..8		Virtual [only] layer 1 offset
/$ctl/layer/VRT/1/name	S		10 chars max … [TBV]	Virtual [only] layer 1 name
/$ctl/layer/VRT/1/1	N	1..16		Virtual [only] layer 1, node 1
/$ctl/layer/VRT/1/1/type	S		OFF, CH, BUS, DCA, MIDI, SEND, FX, SENDS	Virtual [only] layer 1, node 1 type (OSC patterns in italic below correspond to MIDI type)
/$ctl/layer/VRT/1/1/i	I	1..127		Virtual [only] layer 1, node 1 index
/$ctl/layer/VRT/1/1/dst	I	1..28		Virtual [only] layer 1, node 1 destination index (used for type SEND)

[71] Can be set, but will likely be replaced by default names from console layer changes

[72] VRT console layers names [TBV]

/$ctl/layer/VRT/1/1/val	*I*	*0..127*		Virtual [only] *layer 1, node 1 value (when type MIDI)*
/$ctl/layer/VRT/$spill				Virtual WING [only] $spill
/$ctl/layer/VRT/$spill/ofs	I	0..64		Virtual WING [only] $spill offset
/$ctl/layer/VRT/$spill/name	S		10 chars max	Virtual WING [only] layer $spill name[73]
/$ctl/layer/VRT/$spill/1	N	1..76		Virtual WING [only] layer $spill, node 1
/$ctl/layer/VRT/$spill/1/type	S		OFF, CH, BUS, DCA, MIDI, SEND, FX, SENDS	Virtual WING [only] layer $spill, node 1 type (OSC patterns in italic below correspond to MIDI type)
/$ctl/layer/VRT/$spill1/i	I	1..127		Virtual WING [only] layer $spill, node 1 index
/$ctl/layer/VRT/$spill/1/dst	I	1..28		Virtual WING [only] layer $spill, node 1 destination index (used for type SEND)
/$ctl/layer/*VRT*/$spill/1/val	*I*	*0..127*		Virtual WING [only] *layer $spill, node 1 value (when type MIDI)*
/$ctl/user	N			User node
/$ctl/user/sel	I	1..16		User select[74]
/$ctl/user/mode	S		USER, 2TRK, WLIVE, MGRP, SHOW	User button mode (5 buttons above the wheel on the full-size console)
/$ctl/user/cmode	S		HA, GATE, COMP, FLT, U1, U2, U3, PAN	User channel mode (8 buttons top right corner of the full-size console)
/$ctl/user/gpio	N			User GPIO node
/$ctl/user/gpio/1	N	1..4		User GPIO 1 node
/$ctl/user/gpio/1/bu	N			User GPIO 1 up node
/$ctl/user/gpio/1/bu/mode	S		OFF, MUTE, SOLO, INS1, INS2, MGRP, DCAMUTE, SOF, SPILL, FXPAR, DAWBTN, DAWENC, CHPAGE, PAGE, FDRPAGE, VIEWPAGE, OTHER, GPIO, FSTART, SHOWCTL, SCENES, MIDICCT, MIDICCP, MIDINT, MIDINP, MIDIPGM, USBPR, SDRECA, SESSIONA, MARKERA, SDRECB, SESSIONB, MARKERB	User GPIO 1 up function (see appendix on buttons)
/$ctl/user/gpio/1/bu/name	S		16 chars max	User GPIO 1 up name (use a leading '|' to invert characters)
/$ctl/user/gpio/1/bu/$fname	S		16 chars max	User GPIO 1 up $fname [RO]
/$ctl/user/user	N			User Layer node (bottom with Link enabled)
/$ctl/user/user/1	N	1..4		User layer button 1 node
/$ctl/user/user/1/bu	N			User layer button 1 upper row node

[73] Can be set, but will likely be replaced by default names from console layer changes

[74] Setting values using the range 1..16, reported int values are in the range 0..15, string values in the range 1..16

/$ctl/user/user/1/bu/mode	S		OFF, MUTE, SOLO, INS1, INS2, MGRP, DCAMUTE, SOF, SPILL, FXPAR, DAWBTN, DAWENC, CHPAGE, PAGE, FDRPAGE, VIEWPAGE, OTHER, GPIO, FSTART, SHOWCTL, SCENES, MIDICCT, MIDICCP, MIDINT, MIDINP, MIDIPGM, USBPR, SDRECA, SESSIONA, MARKERA, SDRECB, SESSIONB, MARKERB	User layer button1 upper row function (see appendix on buttons)
/$ctl/user/user/1/bu/name	S		16 chars max	User layer button 1 upper row name (use a leading '|' to invert characters)
/$ctl/user/user/1/bu/$fname	S		16 chars max	User layer button 1 upper row function name [RO]
/$ctl/user/user/1/bd	N			User layer button 1 lower row node
/$ctl/user/user/1/bd/mode	S		OFF, MUTE, SOLO, INS1, INS2, MGRP, DCAMUTE, SOF, SPILL, FXPAR, DAWBTN, DAWENC, CHPAGE, PAGE, FDRPAGE, VIEWPAGE, OTHER, GPIO, FSTART, SHOWCTL, SCENES, MIDICCT, MIDICCP, MIDINT, MIDINP, MIDIPGM, USBPR, SDRECA, SESSIONA, MARKERA, SDRECB, SESSIONB, MARKERB	User layer button 1 lower row function (see appendix on buttons)
/$ctl/user/user/1/bd/name	S		16 chars max	User layer button 1 lower row name (use a leading '|' to invert characters)
/$ctl/user/user/1/bd/$fname	S		16 chars max	User layer button 1 lower row function name [RO]
/$ctl/user/daw1	N			User DAW1 node
/$ctl/user/daw1/1	N	1..4		User DAW1 button 1 node
/$ctl/user/daw1/1/bu	N			User DAW1 button 1 upper row node
/$ctl/user/daw1/1/bu/mode	S		OFF, MUTE, SOLO, INS1, INS2, MGRP, DCAMUTE, SOF, SPILL, FXPAR, DAWBTN, DAWENC, CHPAGE, PAGE, FDRPAGE, VIEWPAGE, OTHER, GPIO, FSTART, SHOWCTL, SCENES, MIDICCT, MIDICCP, MIDINT, MIDINP, MIDIPGM, USBPR, SDRECA, SESSIONA, MARKERA, SDRECB, SESSIONB, MARKERB	User DAW1 button 1 upper row function (see appendix on buttons)
/$ctl/user/daw1/1/bu/name	S		16 chars max	User DAW1 button 1 upper row name (use a leading '|' to invert characters)

/$ctl/user/daw1/1/bu/$fname	S		16 chars max	User DAW1 button 1 upper row function name [RO] (see Appendix)
/$ctl/user/daw1/1/bd	N			User DAW1 button 1 lower row node
/$ctl/user/daw1/1/bd/mode	S		OFF, MUTE, SOLO, INS1, INS2, MGRP, DCAMUTE, SOF, SPILL, FXPAR, DAWBTN, DAWENC, CHPAGE, PAGE, FDRPAGE, VIEWPAGE, OTHER, GPIO, FSTART, SHOWCTL, SCENES, MIDICCT, MIDICCP, MIDINT, MIDINP, MIDIPGM, USBPR, SDRECA, SESSIONA, MARKERA, SDRECB, SESSIONB, MARKERB	User DAW1 button 1 lower row function (see appendix on buttons)
/$ctl/user/daw1/1/bd/name	S		16 chars max	User DAW1 button 1 lower row name (use a leading '|' to invert characters)
/$ctl/user/daw1/1/bd/$fname	S		16 chars max	User DAW1 button 1 lower row function name [RO] (see Appendix)
/$ctl/user/daw2	N			User DAW2 node
/$ctl/user/daw2/1	N	1..4		User DAW2 button 1 node
/$ctl/user/daw2/1/bu	N			User DAW2 button 1 upper row node
/$ctl/user/daw2/1/bu/mode	S		OFF, MUTE, SOLO, INS1, INS2, MGRP, DCAMUTE, SOF, SPILL, FXPAR, DAWBTN, DAWENC, CHPAGE, PAGE, FDRPAGE, VIEWPAGE, OTHER, GPIO, FSTART, SHOWCTL, SCENES, MIDICCT, MIDICCP, MIDINT, MIDINP, MIDIPGM, USBPR, SDRECA, SESSIONA, MARKERA, SDRECB, SESSIONB, MARKERB	User DAW2 button 1 upper row function (see appendix on buttons)
/$ctl/user/daw2/1/bu/name	S		16 chars max	User DAW2 button 1 upper row name (use a leading '|' to invert characters)
/$ctl/user/daw2/1/bu/$fname	S		16 chars max	User DAW2 button 1 upper row function name [RO] (see Appendix)
/$ctl/user/daw2/1/bd	N			User DAW2 button 1 lower row node
/$ctl/user/daw2/1/bd/mode	S		OFF, MUTE, SOLO, INS1, INS2, MGRP, DCAMUTE, SOF, SPILL, FXPAR, DAWBTN, DAWENC, CHPAGE, PAGE, FDRPAGE, VIEWPAGE, OTHER, GPIO, FSTART, SHOWCTL, SCENES, MIDICCT, MIDICCP, MIDINT, MIDINP, MIDIPGM, USBPR, SDRECA, SESSIONA,	User DAW2 button 1 lower row function (see appendix on buttons)

			MARKERA, SDRECB, SESSIONB, MARKERB	
/$ctl/user/daw2/1/bd/name	S		16 chars max	User DAW2 button 1 lower row name (use a leading '|' to invert characters)
/$ctl/user/daw2/1/bd/$fname	S		16 chars max	User DAW2 button 1 lower row function name [RO] (see Appendix)
/$ctl/user/daw3	N			User DAW3 node
/$ctl/user/daw3/1	N	1..4		User DAW3 button 1 node
/$ctl/user/daw3/1/bu	N			User DAW3 button 1 upper row node
/$ctl/user/daw3/1/bu/mode	S		OFF, MUTE, SOLO, INS1, INS2, MGRP, DCAMUTE, SOF, SPILL, FXPAR, DAWBTN, DAWENC, CHPAGE, PAGE, FDRPAGE, VIEWPAGE, OTHER, GPIO, FSTART, SHOWCTL, SCENES, MIDICCT, MIDICCP, MIDINT, MIDINP, MIDIPGM, USBPR, SDRECA, SESSIONA, MARKERA, SDRECB, SESSIONB, MARKERB	User DAW3 button 1 upper row function (see appendix on buttons)
/$ctl/user/daw3/1/bu/name	S		16 chars max	User DAW3 button 1 upper row name (use a leading '|' to invert characters)
/$ctl/user/daw3/1/bu/$fname	S		16 chars max	User DAW3 button 1 upper row function name [RO]
/$ctl/user/daw3/1/bd	N			User DAW3 button 1 lower row node
/$ctl/user/daw3/1/bd/mode	S		OFF, MUTE, SOLO, INS1, INS2, MGRP, DCAMUTE, SOF, SPILL, FXPAR, DAWBTN, DAWENC, CHPAGE, PAGE, FDRPAGE, VIEWPAGE, OTHER, GPIO, FSTART, SHOWCTL, SCENES, MIDICCT, MIDICCP, MIDINT, MIDINP, MIDIPGM, USBPR, SDRECA, SESSIONA, MARKERA, SDRECB, SESSIONB, MARKERB	User DAW3 button 1 lower row function (see appendix on buttons)
/$ctl/user/daw3/1/bd/name	S		16 chars max	User DAW3 button 1 lower row name (use a leading '|' to invert characters)
/$ctl/user/daw3/1/bd/$fname	S		16 chars max	User DAW3 button 1 lower row function name [RO] (see Appendix)
/$ctl/user/daw4	N			User DAW4 node
/$ctl/user/daw4/1	N			User DAW4 button 1 node
/$ctl/user/daw4/1/bu	N	1..4		User DAW4 button 1 upper row node
/$ctl/user/daw4/1/bu/mode	S		OFF, MUTE, SOLO, INS1, INS2, MGRP, DCAMUTE, SOF, SPILL, FXPAR, DAWBTN, DAWENC, CHPAGE, PAGE, FDRPAGE,	User DAW4 button 1 upper row function (see appendix on buttons)

			VIEWPAGE, OTHER, GPIO, FSTART, SHOWCTL, SCENES, MIDICCT, MIDICCP, MIDINT, MIDINP, MIDIPGM, USBPR, SDRECA, SESSIONA, MARKERA, SDRECB, SESSIONB, MARKERB	
/$ctl/user/daw4/1/bu/name	S		16 chars max	User DAW4 button 1 upper row name (use a leading '|' to invert characters)
/$ctl/user/daw4/1/bu/$fname	S		16 chars max	User DAW4 button 1 upper row function name [RO] (see Appendix)
/$ctl/user/daw4/1/bd	N			User DAW4 button 1 lower row node
/$ctl/user/daw4/1/bd/mode	S		OFF, MUTE, SOLO, INS1, INS2, MGRP, DCAMUTE, SOF, SPILL, FXPAR, DAWBTN, DAWENC, CHPAGE, PAGE, FDRPAGE, VIEWPAGE, OTHER, GPIO, FSTART, SHOWCTL, SCENES, MIDICCT, MIDICCP, MIDINT, MIDINP, MIDIPGM, USBPR, SDRECA, SESSIONA, MARKERA, SDRECB, SESSIONB, MARKERB	User DAW4 button 1 lower row function (see appendix on buttons)
/$ctl/user/daw4/1/bd/name	S		16 chars max	User DAW4 button 1 lower row name (use a leading '|' to invert characters)
/$ctl/user/daw4/1/bd/$fname	S		16 chars max	User DAW4 button 1 lower row function name [RO] (see Appendix)
/$ctl/user/1	N	1..16, U1..U4, MM, D1..D4		User 1 node[75]
/$ctl/user/1/1	N	1..4		User 1 button/encoder 1 node
/$ctl/user/1/1/led	I	0..1		User 1 LED 1 off/on switch
/$ctl/user/1/1/col	I	1..18		User 1 LED 1 color
/$ctl/user/1/1/enc	N			User 1 encoder 1 node[76]
/$ctl/user/1/1/enc/mode	S		OFF, FDR, PAN, DCA, SSND, FSND, FX, DAWMCU, MON, OTHER, MIDICC, SD A, SD B	User 1 encoder 1 function (see appendix on buttons)
/$ctl/user/1/1/enc/name	S		16 chars max	User 1 encoder 1 name (use a leading '|' to invert characters, use a leading '*' to ensure value is displayed, or '|*' for both actions)
/$ctl/user/1/1/enc/$fname	S		16 chars max	User 1 encoder 1 function name [RO]
/$ctl/user/1/1/bu	N			User 1 button 1 upper row node

[75] Some `/$ctl/user/xx` values [i.e U1..4, MM, D1..4] are specific to Compact (U1..4: User, MM: Main/Matrix, D1..4: DCA)

[76] Not valid on Compact models

/$ctl/user/1/1/bu/mode	S		OFF, MUTE, SOLO, INS1, INS2, MGRP, DCAMUTE, SOF, SPILL, FXPAR, DAWBTN, DAWENC, CHPAGE, PAGE, FDRPAGE, VIEWPAGE, OTHER, GPIO, FSTART, SHOWCTL, SCENES, MIDICCT, MIDICCP, MIDINT, MIDINP, MIDIPGM, USBPR, SDRECA, SESSIONA, MARKERA, SDRECB, SESSIONB, MARKERB	User 1 button 1 upper row function (see appendix on buttons)
/$ctl/user/1/1/bu/name	S		16 chars max	User 1 button 1 upper row name (use a leading bu/mode'|' to invert characters)
/$ctl/user/1/1/bu/$fname	S		16 chars max	User 1 button 1 upper row function name [RO]
/$ctl/user/1/1/bd	N			User 1 button 1 lower row node[77]
/$ctl/user/1/1/bd/mode	S		OFF, MUTE, SOLO, INS1, INS2, MGRP, DCAMUTE, SOF, SPILL, FXPAR, DAWBTN, DAWENC, CHPAGE, PAGE, FDRPAGE, VIEWPAGE, OTHER, GPIO, FSTART, SHOWCTL, SCENES, MIDICCT, MIDICCP, MIDINT, MIDINP, MIDIPGM, USBPR, SDRECA, SESSIONA, MARKERA, SDRECB, SESSIONB, MARKERB	User 1 button 1 lower row function (see appendix on buttons)
/$ctl/user/1/1/bd/name	S		16 chars max	User 1 button 1 lower row name (use a leading '|' to invert characters)
/$ctl/user/1/1/bd/$fname	S		16 chars max	User 1 button 1 lower row function name [RO]
/$ctl/user/cuser	N	1..3		Cuser node

/$ctl/user/cuser/1	N		1, 2, 3, … , 23, 24	Cuser 1 rotary knob position; Keeps the bus send value assigned to respective channel for the F1..F3 section. Can be set/changed by holding the F1..F3 key and turning the knob.
				`1..16` maps to `SEND 1..16`
				`17..24` maps to `SEND MX 1..8`

/$ctl/gpio	N			GPIO node
/$ctl/gpio/1	N	1..4		GPIO 1 node
/$ctl/gpio/1/mode	S		TGLNO, TGLNC, INNO, INNC, OUTNO, OUTNC	GPIO 1 mode (TGL: toggle; NO: normally open; NC: normally closed)
/$ctl/gpio/1/$state	I	0..1		GPIO 1 state [RO]
/$ctl/gpio/1/gpstate	I	0..1		GPIO 1 gpio state

[77] Not valid on Compact models

/$ctl/safes	N			Global Safes node
/$ctl/safes/ch	S		40 chars max	Ch safes switches (+ or space)
/$ctl/safes/aux	S		8 chars max	Aux safes switches (+ or space)
/$ctl/safes/bus	S		16 chars max	Bus safes switches (+ or space)
/$ctl/safes/main	S		4 chars max	Main safes switches (+ or space)
/$ctl/safes/mtx	S		8 chars max	Matrix safes switches (+ or space)
/$ctl/safes/dca	S		16 chars max	DCA safes switches (+ or space)
/$ctl/safes/mute	S		8 chars max	Mute safes switches (+ or space)
/$ctl/safes/fx	S		16 chars max	FX safes switches (+ or space)
/$ctl/safes/source	N			Source Safes node
/$ctl/safes/source/LCL	S		24 chars max	LCL source safes switches (+ or space)
/$ctl/safes/source/AUX	S		8 chars max	AUX source safes switches (+ or space)
/$ctl/safes/source/A	S		48 chars max	A source safes switches (+ or space)
/$ctl/safes/source/B	S		48 chars max	B source safes switches (+ or space)
/$ctl/safes/source/C	S		48 chars max	C source safes switches (+ or space)
/$ctl/safes/source/SC	S		32 chars max	SC source safes switches (+ or space)
/$ctl/safes/source/USB	S		48 chars max	USB source safes switches (+ or space)
/$ctl/safes/source/CRD	S		64 chars max	CRD source safes switches (+ or space)
/$ctl/safes/source/MOD	S		64 chars max	MOD source safes switches (+ or space)
/$ctl/safes/source/PLAY	S		4 chars max	REC source safes switches (+ or space)
/$ctl/safes/source/AES	S		2 chars max	AES source safes switches (+ or space)
/$ctl/safes/source/USR	S		48 chars max	USR source safes switches (+ or space)
/$ctl/safes/source/OSC	S		2 chars max	Osc source safes switches (+ or space)
/$ctl/safes/output	N			Output Safes node
/$ctl/safes/output/LCL	S		8 chars max	LCL out safes switches (+ or space)
/$ctl/safes/output/AUX	S		8 chars max	AUX out safes switches (+ or space)
/$ctl/safes/outpu/A	S		48 chars max	A out safes switches (+ or space)
/$ctl/safes/output/B	S		48 chars max	B out safes switches (+ or space)
/$ctl/safes/output/C	S		48 chars max	C out safes switches (+ or space)
/$ctl/safes/output/SC	S		32 chars max	SC out safes switches (+ or space)
/$ctl/safes/output/USB	S		48 chars max	USB out safes switches (+ or space)
/$ctl/safes/output/CRD	S		64 chars max	CRD out safes switches (+ or space)
/$ctl/safes/output/MOD	S		64 chars max	MOD out safes switches (+ or space)
/$ctl/safes/output/REC	S		4 chars max	REC out safes switches (+ or space)
/$ctl/safes/output/AES	S		2 chars max	AES out safes switches (+ or space)
/$ctl/safes/area	N			Area safes node
/$ctl/safes/area/LEFT	S		7 chars max	Left area safes switches (+ or space)
/$ctl/safes/area/CENTER	S		6 chars max	Center area safes switches (+ or space)
/$ctl/safes /area/RIGHT	S		7 chars max	Right area safes switches (+ or space)
/$ctl/safes/area/COMPACT	S		9 chars max	Compact area safes switches (+ or space)
/$ctl/safes/area/RACK	S		5 chars max	Rack area safes switches (+ or space)
/$ctl/safes /area/EXTERN	S		8 chars max	Extern area safes switches (+ or space)
/$ctl/safes /area/VIRTUAL	S		8 chars max	Virtual area safes switches (+ or space)
/$ctl/safes/custom	S		31 chars max	Custom area safes switches (+ or space)
/$ctl/safes/setup	S		3 chars max	Setup area safes switches (+ or space)
/$ctl/daw	N			DAW node

/$ctl/daw/on	I	0..1		DAW enable
/$ctl/daw/conn	S		DIN, USB	DAW connection
/$ctl/daw/emul	S		MCU, HUI	DAW emulation
/$ctl/daw/config	S		CC, MSTR, MSTR1EXT, MSTR2EXT	DAW configuration
/$ctl/daw/ccup	I	0..1		DAW use upper cc
/$ctl/daw/disjog	I	0..1		DAW disable wheel during play
/$ctl/daw/preset	S		-, cubase, live, logicx, nuendo, protools, reaper, studioone	DAW last loaded preset
/$ctl/daw /$on	I	0..1		DAW enable switch
/$ctl/daw/$bpage	I	0..4		DAW on button page
/$ctl/daw/$btntouch	I	0..1		DAW on button sel fader touch
/$ctl/daw/$btnvpot	I	0..1		DAW on button sel vpot
/$ctl/daw/$btnrecrdy	I	0..1		DAW on button sel record ready
/$ctl/daw/$btnauto	I	0..1		DAW on button sel auto
/$ctl/daw/$btnvsel	I	0..1		DAW on button sel v-sel
/$ctl/daw/$btninsert	I	0..1		DAW on button sel insert
/$ctl/midi	N			Midi node
/$ctl/midi/enchctl	S		OFF, DIN, USB	Channel control (FDR, MUTE, PAN)
/$ctl/midi/enfxctl	S		OFF, DIN, USB	FX parameter control
/$ctl/midi/encustctl	S		OFF, DIN, USB	Custom control (RX only)
/$ctl/midi/ensysex	S		OFF, DIN, USB	SYSEX control
/$ctl/midi/enmidicc	S		OFF, DIN, USB	External MIDI control
/$ctl/midi/enscenes	S		OFF, DIN, USB	Scene change
/$ctl/midi/enshowctl	S		OFF, DIN, USB	Show control
/$ctl/midi/enscenetx	S		OFF, DIN, USB	Scene MIDI TX
/$ctl/OSC	N			OSC node
/$ctl/OSC/ronly	I	0..1		Console OSC read only switch
/$ctl/lib	N			Library node (Shows, Scenes, Snaps, …)
/$ctl/lib/$scenes	S		Ex: `scene_1, scene_2, scene_3`	List of Scene[78] names [RO] [79] in the currently opened show
/$ctl/lib/$actidx	I	0..n		Scene number currently loaded/active [RO][80]
/$ctl/lib/$active	S		256 chars max	Name of the active scene [RO], ex: `I:SHOW2/scene_1.snap`
/$ctl/lib/$actshow	S		256 chars max	Name of the active show [RO], after having pressed on the "OPEN SHOW" icon, ex: `I:SHOW2`
/$ctl/lib/$action	S		IDLE, GOPREV, GONEXT, GO, PREV, NEXT, GOTAG	Show control actions, after having pressed on the "OPEN SHOW" icon[81]
/$ctl/lib/$actionidx	I	0..16384		Scene number user selection, in the list of Scenes. A show must be opened.

[78] A Scene is a Snap, a Snippet, a Preset, an Audio clip, or a combination thereof, referenced in a Show file

[79] Only the first name of the list is returned by std OSC command. You must use the node definition command (OSC or native interface). To get the full contents, for ex: `/$ctl/lib~~~,s~~?~~~.` A show must be opened for the command to be active

[80] A show must be opened for the command to be active. 0 means "no show Scene loaded"

[81] Make sure `/$ctl/lib/$actionidx` is set to `0` if you want to use `NEXT` or `PREV` followed by a `GO`.

				Use `/$ctl/lib/$action ,s GO` to load the Scene entity `$actionidx` points to, or `GOTAG` if `$actionidx` refers to the tag number of a tagged scene in the show. `$actionidx` should be 0 to enable **NEXT** or **PREV** to work as expected with a **GO** command.
/$ctl/lib/$activeid	I	0..128		Tag of the active scene if present [RO]
/$ctl/$globals	N			CTL Global Settings node
/$ctl/$globals/fdrsel	I	0..1		Screen Touch Fader Select
/$ctl/$globals/fdrres	S		NORM, FINE, AUTO	Fader resolution
/$ctl/$globals/fdrspd	S		SLOW, MED, FAST	Fader speed
/$ctl/$globals/mousetchdis	I	0..1		Mouse disable touch
/$ctl/$globals/mousespd	F	0.1..2.0	191 steps	Mouse speed
/$ctl/$globals/tapflash	S		OFF, 8X, ON	Tap Tempo Flash
/$ctl/$globals/srcdisp	I	0..1		Show source on scribble
/$ctl/$globals/lockmtr	I	0..1		Show meter page when locked
/$ctl/$globals/showscene	I	0..1		Always show active scene
/$ctl/$globals/cf_load	I	0..1		Confirm Snapshot Load
/$ctl/$globals/cf_upd	I	0..1		Confirm Snapshot Update
/$ctl/$globals/usewheel	I	0..1		Use wheel to navigate in lists of items (snaps, files,…)
/$ctl/$globals/timefmt	S		24H, 12H	Time format
/$ctl/$globals/date fmt	S		YMD, DMY	Date format
/$ctl/$globals/$filesort	S		NAME, TYPE, 0->9, 9->0	File sort order
/$ctl/$globals/$noautosave	I	0..1		Auto save switch (0=autosave)
/$ctl/$globals/$savenow	I	0..1		Save console data now[82]

[82] This command should not be used as part of a program loop as it will eventually wear the flash memory where data is saved. This command may change in the future.

Global Settings

Command	Type	Range	Text	Description
/$globals	N			Global Settings node
/$globals/clkrate	S		48000.0, 44100.0	Master clock rate
/$globals/clksrc	S		INT, A, B, C, AES, CARD, MOD	Master clock source
/$globals/startmute	I	0..1		Mute outputs on startup
/$globals/usbacfg	S		2/2, 8/8, 16/16, 32/32, 48/48	USB Input/Output configuration
/$globals/sccfg	S		AUTO, 0/32, 1/31, 2/30, 3/29, 4/28, 5/27, 6/26, 7/25, 8/24, 9/23, 10/22, 11/21, 12/20, 13/19, 14/18, 15/17, 16/16, 17/15, 18/14, 19/13, 20/12, 21/11, 22/10, 23/9, 24/8, 25/7, 26/6, 27/5, 28/4, 29/3, 30/2, 31/1, 32/0	SC Configuration
/$globals/harmt	N			HA remote node
/$globals/harmt/a	I	0..1		Enable HA remote on AES-A
/$globals/harmt/b	I	0..1		Enable HA remote on AES-B
/$globals/harmt/c	I	0..1		Enable HA remote on AES-C
/$globals/custsync	N			Custom Sync node
/$globals/custsync /a	I	0..1		Enable Cust Sync on AES-A
/$globals/custsync /b	I	0..1		Enable Cust Sync on AES-B
/$globals/custsync /c	I	0..1		Enable Cust Sync on AES-C

WING native / binary data interface

WING native / binary data interface

WING exposes a binary structure that contains all data presented above in the WING snapshot JSON structure chapter, and more objects. Some objects are `READ ONLY` while others can bet accessible for `get()` and `set()` functions. One can access the object data, effectively getting access to the value assigned to the object, or to the object description, a special class that provides the object name, type, min/max values, or enumerated values.
As mentioned before, communication takes place using `TCP`. We give all the basic/necessary details for communicating with WING below; this data is coming from Behringer.
Applications can communicate with the console using `TCP` port `2222`. The console will reject further connection requests, if the maximum number of simultaneous connections (currently `16`) is reached. Open connections will time out after `10 seconds` of inactivity (on the receiving side).

Communication Channels

Communication uses 14 distinct *channels* [1..14] corresponding to the Channel IDs presented below

Table 1. Channel Usage

channel	ChID	Usage
1	0	n/u
2	1	Audio Engine & Control requests
3	2	n/u
4	3	Meter Data Requests
5	4	n/u
6	5	n/u
7	6	n/u
8	7	n/u
9	8	n/u
10	9	n/u
11	A	n/u
12	B	n/u
13	C	n/u
14	D	n/u

To select/change the active channel, use the following sequence
`0xdf, 0xd<ChID>`

When communicating with WING, the escape byte `0xdf` should be handled carefully, as shown in the two routines shown below for sending and receiving data.

Sample receive routine

```
#define NRP_ESCAPE_CODE         0xdf
#define NRP_CHANNEL_ID_BASE     0xd0
#define NRP_NUM_CHANNELS        14

void Cnrpclientconnector::nrpc_data_rx(byte db)
{
  if (db == NRP_ESCAPE_CODE && !escf) escf = true;
  else {
    if (escf) {
      if (db != NRP_ESCAPE_CODE) {
        escf = false;
        if (db == NRP_ESCAPE_CODE - 1) db = NRP_ESCAPE_CODE;
        else if (db >= NRP_CHANNEL_ID_BASE &&
                       db < NRP_CHANNEL_ID_BASE + NRP_NUM_CHANNELS) {
          if (ch_id_rx != db - NRP_CHANNEL_ID_BASE) {
            ch_id_rx = db - NRP_CHANNEL_ID_BASE;
          }
          return;
        } else if (ch_id_rx >= 0) data_rx(ch_id_rx, NRP_ESCAPE_CODE);
      }
    }
    if (ch_id_rx >= 0) data_rx(ch_id_rx, db);
  }
}
```

Example: The sequence D702DFDEAF0E02 will in fact represent D702DFAF0E02

Sample transmit routine

```
void Cnrpclientconnector::data_tx(int ch_id, const void* data, int len)
{
  assert(ch_id >= 1 && ch_id <= NRP_NUM_CHANNELS);

  if (ch_id_tx != ch_id) {
    ch_id_tx = ch_id;
    nrpc_data_tx_flush();
    nrpc_data_tx(NRP_ESCAPE_CODE);
    nrpc_data_tx(ch_id + NRP_CHANNEL_ID_BASE);
  }

  bool esc = false;
  byte* dpp = (byte*)data;
  while (len-- > 0) {
    byte db = *dpp++;
    if (db == NRP_ESCAPE_CODE) esc = true;
    else {
      if (esc && db >= NRP_CHANNEL_ID_BASE &&
                  db <= NRP_CHANNEL_ID_BASE + NRP_NUM_CHANNELS) {
        db = NRP_ESCAPE_CODE - 1;
        dpp--;
        len++;
      }
      esc = false;
    }
    nrpc_data_tx(db);
  }
  if (esc) nrpc_data_tx(NRP_ESCAPE_CODE - 1);
}
```

Examples: With current tx channel being 3, sending `d702dfaf0e02` to channel 1 will transfer `dfd0d702dfaf0e02`, sending `d702dfdf0e02` to channel 2 will transfer `dfd1d702dfdf0e02`, and sending `d702dfd10e02` to channel 1 again will transfer `dfd0d702dfded10e02`

Channel 2: Audio Engine

Communication with the audio engine uses a token/data based binary stream protocol. Words are 2 bytes (big endian), longs are 4 bytes (big endian).

Binary Stream Format

Table 2. Binary Stream Protocol Tokens

Token	Data	Function
0x00		false; off; 0
0x01		true; on; 1
0x02…0x3f		int 2…63
0x40…0x7f		node index 1…64
0x80…0xbf		string[1…64]
0xc0…0xcf		node name[1…16]
0xd0		empty string
0xd1	byte	string[1…256]
0xd2	word	node index 1…65536
0xd3	word	int16
0xd4	long	int32
0xd5	long	float32
0xd6	long	raw float32 (0.0…1.0)
0xd7	long	node hash
0xd8		click (toggle)
0xd9	byte	step (inc/dec)
0xda		node tree: goto root node
0xdb		node tree: 1 level up
0xdc		data request
0xdd		request node definition (current node)
0xde		end of data/def request
0xdf	word	node definition response (word: data length in bytes)
0xe0…0xff		not used

Navigation within the nodes tree can be done by using `root`, `1 level up`, `node index`, `_node name`, or `node hash` tokens.
Definitions of all `sub-nodes` within the current `node` can be requested with the `request node definition` token.

Node Definition Response

```
0xdf, len.w, [len.l], parent.l, hash.l, index.w, namelen.b, name, longnamelen.b, longname, flags.w,
        (If len.w==0 then len.l is used)
        node:   -
        linf:   min.f, max.f, steps.l
        logf:   min.f, max.f, steps.l
        fdr:    -
        int:    min.l, max.l
        enum:   count.w, [itemlen.b, item, longitemlen.b, longitem] * count
        fenum:  count.w, [itemval.f, longitemlen.b, longitem] * count
        string: maxlen.w
```

Where the `node type` can be derived from the `flags` as follows:

Table 3. Node Flags

Flags	Function	Details
b0…b3	unit	0: none 1: dB 2: % 3: ms 4: Hz 5: mtrs

		6: seconds 7: octaves
b4…b7	type	0: node 1: lin float 2: log float 3: fader level 4: integer 5: string enum 6: float enum 7: string
b9	r/o	read-only flag

Channel 3: Metering

Use this channel to request metering data from the console. The data is sent back to the client `IP` to the specified `port`. Meter data times out after `5` seconds.
Each metering data block is prefixed by a client-specified 4byte `report id`.
To avoid confusion on the receiver side, the client should use different `report id` values for different meter collections.
Meter collections are specified by using multiple type/index elements. Specified meters are sent back in the sequence corresponding to the request message.

Meter values are coded on `2 bytes` (signed, big endian). Level values are in `1/256` dB.

Most data are returned in `1/256` steps. This is typically the case for `Gate gain` and `Dyn gain` values. The returned value is multiplied/adjusted with a fixed value to cover for the plugin model data range in use. For most of them `1.0 (256)` maps to `20` dB gain reduction. Standard Wing gate is `60` dB.

`fx` meter subscriptions return 4 levels and 6 state meters (see table below: Meter Data). Band gain reduction is in the first 5 of these. `Value` in dB = return value `* 6.0 / 2048`.

`Gate LED` and `Dyn State` in * V2 subscriptions return a value of 0 or 1, depending on the state of the respective channel's Gate and Dyn LEDs, respectively.

Meter Request Tokens

Table 4. Meter Request Tokens

Token	Data	Function
0xd3	word	client *UDP* port (2 bytes big endian; set at least once)
0xd4	long	report id (repeat this token with current id to reset transmission timeout)
0xdc		start of meter collection definition
	0xa0	channel (1…40)
	0xa1	aux (1…8)
	0xa2	bus (1…16)
	0xa3	main (1…4)
	0xa4	matrix (1…8)
	0xa5	dca (1…8)
	0xa6	fx processor (1…16)
	0xa7	source (input) device (1…16)
	0xa8	output device (1…11)
	0xa9	monitor (no index)
	0xaa	rta (no index)
	0xab	channel V2 (1…40)
	0xac	aux V2 (1…8)
	0xad	bus V2 (1…16)
	0xae	main V2 (1…4)
	0xaf	matrix V2 (1…8)
	0x00…0x7f	index 1…128 (can be repeated multiple times)
0xde		end of meter collection definition

Example specification

```
0xdc 0xa0 0x00 0x01 0x08 0xa6 0x04 0xde
```

Requests meter data for channels strips 1,2,9, and fx 5.

Meter Data Packet Structure := <report id (4 bytes)><Meter Data (n words)>

Meter Data

Table 5. Meter Data	
Section	Contents
channel aux bus main matrix	input left input right output left output right gate key gate gain dyn key dyn gain
dca	pre fader left pre fader right post fader left post fader right
fx	input left input right output left output right state meters (1…6)
source	source group levels (i.e. local ins: 8 meters)
output	output group levels (i.e. local outs: 8 meters)
monitor	solo bus left solo bus right mon 1 left mon 1 right mon 2 left mon 2 right
rta	rta slot meters (120)
Channel V2 aux V2 bus V2 main V2 matrix V2	input left input right output left output right gate key gate gain gate led dyn key dyn gain dyn state automix gain

We show below a typical binary communication sequence for requesting metering data:

```
→W  5 B: dfd3d33737                         // Declaring port to use is 0x3737 = 14135
→W 11 B: dfd3d400000002dca001de             // Meters request id = 2, channel 02
→W  7 B: dfd3d400000002                     // Renew request id = 2, meter data for 5s
```

Introducing wapi [wapi]

The previous chapters on `JSON` structures and binary, token-based communication may not be very accessible to many programmers. For that reason, a more accessible API (Application Programming Interface) available as a set of include files and libraries is proposed here. It is written in C, which ensure it can easily be used in many applications, providing good performance.

There are two include files that should always be part of your programs as they contain information about the `JSON` structure and the objects respective binary pointers in WING. The API provides an abstraction layer to the binary interface and procedure calls for standard functions to get, set, manipulate WING data.

At the API level, WING data can be 32bit int, 32bit float or string data. All API data in little-endian, enabling easy use in standard programming languages.

Besides this document, the wapi API consists of two include files and a library:

`wapi.h` is the main include file containing enumerated types for errors, token types, and wapi abstraction enumerated tokens.

`wext.h` is a file containing the definitions of external library calls to wapi.lib the actual library of API functions.

`wapi.lib` is a static-link library for linking with your application. The library contains all wapi functions that can be used and are described later in this document.

A typical program accessing WING starts with an '*open*' function and ends with a '*close*' function. These two functions establish the communication path to WING on your local network and ensure data is properly cleaned when leaving the program.

Programs communicate with WING over network. The API call `wOpen()` is used to establish communication link between WING and the application.

WING supports multiple formats, including integers, floats, and strings types. The API will try to ensure conversions as best as possible in order to match the requested format either by WING or by the API command. For example, if you request float data from a WING token which is an integer, the API will convert the integer to float before returning the data. Similarly, if you set a WING token of type string by sending it a float value, the float data will be changed to string before being sent to WING.

wapi tokens

`wapi` makes use of `tokens` (an enumerated 32bits integer acting as a unique identifier) to identify the subtrees and leaves of the WING `JSON` hierarchy structure. WING tokens are easily identified by their name, based on their corresponding `JSON` structure name taken from the WING hierarchical data tree we already presented in this document.

For example the identifier for "channel 1 mute control", a.k.a. "`ch.1.mute`" in the `JSON` tree is known as token `CH_1_MUTE`. The respective parameter in WING internal data structure is an integer that can be 0 or 1 and as written above, can be modified (or set) from integer, float or even string data types, and can be returned to the application also as an integer, a float or string.
Following the naming convention above "channel 1 fader value" will be identified as "`CH_1_FDR`", "bus 14 panoramic value" will be identify as "`BUS_14_PAN`", and so on.

Some identifiers can have names that are not as obvious, and this typical of some of the dynamically assignable subtrees of the `JSON` tree. Typically, the filter, gate, compressor, and equalizer of WING channels can be assigned different plugin models, set for example using the `wapi CH_1_EQ_MDL` token (or `OSC ch/1/eq/mdl`), known as channel 1 EQ model in the case of channel 1 and its EQ setting. If you report to the different types (or models) of EQ plugins in the appendices of this document, you will see that the EQ model can be one of several choices, each model having different settings. In fact, every single setting maps to a given token, based on their respective listing number, i.e. `setting #7`, a.k.a. "`1q`" for EQ model "`STD`" will have the same token value as `setting #7` for EQ model `SOUL`, known as "`1mg`".

To enable `wapi` managing these different mapping, all `JSON` "dynamic" parameters are named after their listing number, rather than their name for a given effect or plugin model. As a result, and taking the case of EQ models "`STD`" and "`SOUL`" above, setting "`1q`" and "`1mg`" will have the share the same token ending with "`7`" (for listing #7).

The naming convention above applies to the following:

- Channel: filter, gate, compressor, equalizer ["`1`", "`2`", … "`33`"]
- Bus, Mains, Matrix: compressor, equalizer ["`1`", "`2`", … "`33`"]
- FX: all fx meter settings ["`1`", "`2`", … "`33`"]
- GPIOs ["`1`", "`2`"]
- User buttons ["`1`", "`2`", "`3`"]
- Layered user encoders and buttons ["`1`", "`2`", "`3`"]

Some tokens correspond to read-only data; trying to change their value will result in an error returned to the calling function.

WING tokens are listed in an include file: `wapi.h` that must be included in your program. The `wapi.h` include file also contains the status or error codes that can be returned by API function calls.

Compiling a program using wapi

All function calls are regrouped in a binary library: `wapi.lib`, that you must include at link time.
A typical compilation of a source file `wtest.c` in a Windows environment can be as follows:

```
gcc -O3 -Wall -c -fmessage-length=0 -o wtest.o "wtest.c"
gcc "-LC:<path to wapi.lib>" -o wtest.exe wtest.o -lwapi -lws2_32
```

Don't forget to set the correct path to the `wapi.lib` file in the above compilation/link lines!
Depending on your application, you may need to provide additional Windows dynamic libraries references (i.e. -lgdi32, -lcomdlg32, etc.)

WING parameters can be set (or modified) using the `wSetxxx` API family of calls; Similarly, the parameters can be retrieved from WING using the `wGetxxx` API family of calls. The following pages will present all API functions and will include examples of source code to help you in your first steps with **wapi**.
Additional calls will serve establishing the communication path with the console, and several services and utility functions needed to parse, or help with data.

wapi Reference Guide

wapi Reference Guide

Open and Close

Int wOpen(char wip)*

`wOpen()` initializes global variables for the application and opens the communication with a WING console responding at `IP` address wip.
`wip` is a string containing the console `IP` data in the form "`xxx.xxx.xxx.xxx`"; if the console `IP` address is unknown, `wip` should be an empty string and provide enough characters to store the IP address where WING will be found. The `wOpen()` function will attempt a network broadcast announce on the `/24`[83] of the local network to identify the first WING that will reply on the local network.
Upon successful completion the function will return `WSUCCESS` and if the `wip` parameter was an empty string when calling the function, it will contain the `IP` at which the console was found. Other values can be returned in case of issues or errors reported.
Once connection is established with the console, it will be kept active for about 10 seconds after which the console will close the link. The `wKeepAlive()` function can be called (before the desk closes communication) to extend the link active another 10 seconds.
It must be noted that if a connection is kept active, changes made directly at the console (by moving a fader, or pressing buttons for example) will generate data the application will continuously receive. This can represent a lot of data the application must be ready to accept and manage. It can also be the source of incorrect data returned to `Get` functions and specific care should be taken when developing live or event-driven applications.

void wClose()

`wClose()` ensures data is correctly disposed of when your program ends. It should be the last call before the `return` statement or `exit` call in your application.

unsigned int wVer()

`wVer()` returns the version of the wapi library file being used. The returned version is in the form 'major.minor.revision-update', and its value is provided as `0xMMmmVVuu`, with `MM.mm` being the standard major.minor version number corresponding as close as possible to the Wing FW release wapi was based on, and `VV-uu` represents a software build revision number and update within MM.mm.

[83] For example, 198.51.100.0/24 is the prefix of the Internet Protocol version 4 network starting at the given address, having 24 bits allocated for the network prefix, and the remaining 8 bits reserved for host addressing. Addresses in the range 198.51.100.0 to 198.51.100.255 belong to this network.

Setting Values

int wSetTokenFloat(wtoken token, float fval)

The `wSetTokenFloat()` sets WING token `token` to float value `fval`. If the token `token` is of a different type than float, `fval` will be adapted to the format expected by token `token`; if token `token` corresponds to a dynamic parameter named `1` to `32`, no format change will take place and the function will set token `token` using `fval` as float.
For example, sending value `444.0` to WING token `CH_1_PEQ_1F` will be sent as a 32bit float value. WING will nevertheless adjust it to the nearest valid value of `444.533997`. Sending that same value `444.0` to WING token `CH_1_PEQ_ON` will result in a setting to `1`; Finally, sending value `444.0` to WING token `CH_1_NAME` will change the channel name to `444.00`.
To change the value of `CH_1_EQ_1` to say 'ten' should be sent as float `10.0`, assuming the parameter's value expected type is float.
The function returns `WSUCCESS` if the requested operation was successful, other values can be returned, such as `WZERO` if no suitable format was found for adapting the value of `fval`, or `WSEND_TCP_ERROR` if an error took place while communicating with WING. Attempting to set a value on a token of type `NODE` will return `WNODE`.

int wSetTokenInt(wtoken token, int ival)

The `wSetTokenInt()` sets WING token `token` to int value `ival`. If the token `token` is of a different type than int, `ival` will be adapted to the format expected by token `token`; if token `token` corresponds to a dynamic parameter named `1` to `32`, no format change will take place and the function will set token `token` using `ival` as int.
For example, sending value `444` to WING token `CH_1_PEQ_ON` will result in a setting to `1`; Finally, sending integer value `444` to WING token `CH_1_NAME` will change the channel name to `444`.
To change the value of `CH_1_GATE_5` to say 'one' with `CH_1_GATE_MDL` set to "`9000G`" should be sent as int `1`, as the parameter's value expected type is int.
The function returns `WSUCCESS` if the requested operation was successful, other values can be returned, such as `WZERO` if no suitable format was found for adapting the value of `ival`, or `WSEND_TCP_ERROR` if an error took place while communicating with WING. Attempting to set a value on a token of type `NODE` will return `WNODE`.

int wSetTokenString(wtoken token, char str)*

The `wSetTokenString()` function takes as input a WING token `token` and a string `str`. It sends to WING the value of `str` after it has been adapted to the format expected by the WING token it is sent to.
For example, sending string "`444`" to WING token `CH_1_PEQ_1F` will be sent as a 32bit float value of `444.0`; WING will the adjust it to the nearest valid value of `444.533997`. Sending that same string **"`444`"** to WING token `CH_1_PEQ_ON` will result in a setting to `1`, Finally, sending string "`444`" to WING token `CH_1_NAME` will change the channel name to `444`.
To change the value of `CH_1_GATE_6` to say 'gate' with `CH_1_GATE_MDL` set to "`9000G`" should be sent as "`GATE`", as the parameter's value expected type is string.
The function returns `WSUCCESS` if the requested operation was successful, other values can be returned, such as `WZERO` if no suitable format was found for adapting the string `str`, or `WSEND_TCP_ERROR` if an error took place while communicating with WING. Attempting to set a value on a token of type `NODE` will return `WNODE`.

int wToggleTokenInt(wtoken token)

The `wToggleTokenInt()` function toggles the `0/1` value of WING token `token`. Token `token` must be of type `int`.
This function offers a way to change or update `0/1` parameter values without having to go through a "read/test/set" roundtrip with the console, providing a more efficient communication path.
The function returns `WSUCCESS` if the requested operation was successful, other values can be returned, such as `WTYPE` if the parameter format for token `token` is not of type integer, or `WSEND_TCP_ERROR` if an error took place while communicating with WING. Attempting to set a value on a token of type `NODE` will return `WNODE`.

int wClickTokenByte(wtoken token, char ival)

The `wClickTokenByte()` function increments or decrements the value of token `token` by the amount represebted by signed byte `ival`. Token `token` can be of type `int` or `float`.
The value of byte `ival` is in the range `[-128..+127]`.
This function offers a way to change or update parameter values without having to go through a "read/test/set" roundtrip with the console, providing a more efficient communication path.
The function returns `WSUCCESS` if the requested operation was successful, other values can be returned, such as `WTYPE` if the parameter format for token `token` is not of type integer or float, or `WSEND_TCP_ERROR` if an error took place while communicating with WING. Attempting to set a value on a token of type `NODE` will return `WNODE`.

Getting Values

`wapi` offers several procedure calls to retrieve data from WING; specific datasets can be of use when getting data. These are defined in `wapi.h`:

`wvalue` is a union type definition to enable receiving several types of data in a single 32bits field.

```
typedef union {
    unsigned int       uval;   // unisgned integer type data
    int                ival;   // integert type data
    float              fval;   // float type data
    char*              sval;   // pointer to string
} wvalue;
```

`wtype` is an enumerated list of ints to provide the data type returned in `wvalue`.

```
typedef enum wtype {
    UNKN = -1,
    NODE,       // node type (unsigned int)
    I32,        // int type
    F32,        // float type
    S32,        // string (char*) type
    V32         // an unsigned int or void type to accept all the above
} wtype;
```

`wTV` is a C structure defined in the `wapi.h` file as follows:

```
typedef struct {
    wtoken    token;
    wtype     type;
    union {
        unsigned int udata;
        int          idata;
        float        fdata;
        char*        sdata;
    } d;
} wTV;
```

The filling of the structure is obvious for `int` and `float` data; `string` data sets are dynamically allocated and the pointer of the allocated string is saved in `sdata`; if the string returned from the console is an empty string, no memory allocation takes place and a `NULL` pointer is set in `sdata`. When receiving an element of type `string` with a valid `sdata` pointer, the application is responsible for freeing the allocated memory pointed to by `sdata` after use.

wtype wGetType(wtoken token)

`wGetType()` returns the type associated to the token `token`; the returned value is one of the types listed in the `wtype` list described above.

char wGetName(wtoken token)*

`wGetName()` returns the string `JSON` descriptor corresponding to token `token`. The returned string is part of the constant definitions of `wapi` and cannot be altered by the calling application.
Note that the string "`$$unkown`" (not part of the `JSON` tree leaves) is returned for tokens that are not found.

whash wGetHash(wtoken token)

`wGetHash()` returns the binary descriptor (an unsigned int) corresponding to token `token`. The returned data can be used to identify specific entries in binary maps returned by `wapi` with the `wGetBinaryNode()` call (see later in this document). The value `0` is returned for tokens that are not found.

*int wGetToken(wtoken token, wtype *type, wvalue *value)*

The `wGetToken()` function retrieves data from WING, based on token `token`.

`wGetToken()` is a generic data retrieval call for `wapi`; Other retrieval functions described later in this document may be more tuned to your specific needs.
Depending on user actions, several data can queue in the receiving buffer; The function will use the data provided by the first occurrence of token `token` in the queue.
The data type returned by WING is used to set a more generic type in `type`, which can be one of int, float or string address on 32 bits data.
The actual value corresponding to int and float data is returned in `value`; In the case of string data (char*), either a NULL pointer is returned for no character present, or `value` contains the pointer to a string of characters.
Note that in the case a string is returned by `wGetToken()`, memory for storing the string will have been allocated by the function. It is the responsibility of the calling application to free the allocated memory when no longer needed to avoid application memory leaks.
`wGetToken()` will return `WSUCCESS` if the token token is found in the receiving queue and valid data is returned, `WZERO` if no valid data type is found or if a timeout occurs during receiving data.
`WMEMORY`, `WSENDERROR` or `WRECVERROR` can be returned in specific error cases.

Below is a small program example of using set() and get() calls, the receiving part using the `wGetToken()` function we first show the display obtained from running the program, followed by the program source code;

```
PS C:\Users\patri\eclipse\wtest\release> ./wtest
WING found at IP: 192.168.1.71
Using version 0.10
type = 3, data = RIDE
type = 2, data = -50.000000
type = 2, data = 0.000000
type = 1, data = 20
type = 2, data = 8.000000
type = 2, data = 0.486968
type = 2, data = 6.000000
```

```
#include <stdio.h>
#include <string.h>
//
#include "wapi.h"
#include "wext.h"
//
```

```
int main() {
    int        i;
    char          wingip[24] = "";
    wtype         type;
    wvalue         value;
    //
    if ((i = wOpen(wingip)) != WSUCCESS) exit(1);
    printf("WING found at IP: %s\n", wingip);
    printf("Using version %i.%i\n", wVer()/256, wVer()&15);
    //
    wSetTokenString(CH_1_GATE_MDL, "RIDE");   //Auto Rider Dynamics
    wSetTokenFloat(CH_1_GATE_1, -50.);         // thr
    wSetTokenFloat(CH_1_GATE_2, 0.);         // tgt
    wSetTokenInt(CH_1_GATE_3, 20);           // spd
    wSetTokenFloat(CH_1_GATE_4, 8.);         // ratio
    wSetTokenFloat(CH_1_GATE_5, 0.5);        // hold
    wSetTokenFloat(CH_1_GATE_6, 6.0);        // range
    //
    wGetToken(CH_1_GATE_MDL, &type, &value);
    if (value.sval) {
        printf("type = %i, data = %s\n", type, value.sval);
        free(value.sval);
    } else {
        printf("no data for ch 1 gate model!\n”);
    }
    wGetToken(CH_1_GATE_1, &type, &value);
    printf("type = %i, data = %f\n", type, value.fval);
    wGetToken(CH_1_GATE_2, &type, &value);
    printf("type = %i, data = %f\n", type, value.fval);
    wGetToken(CH_1_GATE_3, &type, &value);
    printf("type = %i, data = %i\n", type, value.ival);
    wGetToken(CH_1_GATE_4, &type, &value);
    printf("type = %i, data = %f\n", type, value.fval);
    wGetToken(CH_1_GATE_5, &type, &value);
    printf("type = %i, data = %f\n", type, value.fval);
    wGetToken(CH_1_GATE_6, &type, &value);
    printf("type = %i, data = %f\n\n", type, value.fval);
    fflush(stdout);
    return(0);
}
```

int wGetTokenFloat(wtoken token, float fval)*

The `wGetTokenFloat()` function interrogates WING token `token` to get its currently associated value.
As it is the case for `wGetToken(), wGetTokenFloat()` will block until a token `token` is encountered in the receiving queue. The received token value has a given native type, and the function will do its best at converting the received data to float format as expected by the `fval` variable.
For example, inquiring WING token `CH_1_PEQ_1F` will return the current value of the token as a float value in `fval`. Inquiring WING token `CH_1_PEQ_ON` will result in a value of `0.0` or `1.0`, depending on the state of the token.
On the other hand, inquiring WING token `CH_1_NAME` will most likely return a value of `0.0` and a status of `WZERO`; In some cases (i.e., you set the name to be string **“123.7”** for example) you may get a valid floating-point value returned.
The function returns `WSUCCESS` if the requested operation was successful, other values can be returned, such as `WZERO` if no suitable format was found for adapting token value to `fval`, or `WSEND_TCP_ERROR` if an error took place while communicating with WING. Attempting to get a value from a token of type `NODE` will return `WNODE`.

int wGetTokenInt(wtoken token, int ival)*

The `wGetTokenInt()` function interrogates WING token `token` to get its currently associated value. As it is the case for `wGetToken(), wGetTokenInt()` will block until a token `token` is encountered in the receiving queue. The received token value has a given native type, and the function will do its best at converting the received data to integer format as expected by the `ival` variable.
For example, inquiring WING token `CH_1_PEQ_1F` will return the current value of the token as an int value in `ival`. Inquiring WING token `CH_1_PEQ_ON` will result in a value of `0` or `1`, depending on the state of the token. On the other hand, inquiring WING token `CH_1_NAME` will return a value of `0` and a status of `WZERO`; In some cases (i.e., you set the name to be string **"12"** for example) you may get a valid int value returned.
The function returns `WSUCCESS` if the requested operation was successful, other values can be returned, such as `WZERO` if no suitable format was found for adapting token value to `ival`, or `WSEND_TCP_ERROR` if an error took place while communicating with WING. Attempting to get a value from a token of type `NODE` will return `WNODE`.

int wGetTokenString(wtoken token, char str)*

The `wGetTokenString()` function interrogates WING token `token` to get its currently associated value. As it is the case for `wGetToken(), wGetTokenString()` will block until a token `token` is encountered in the receiving queue. The received token value has a given native type, and the function will do its best at converting the received data to string/char* format as expected by the `str` variable.
For example, inquiring WING token `CH_1_PEQ_1F` will return the current value of the token as a string in `str`. Inquiring WING token `CH_1_PEQ_ON` will result in a 1-character string of "`0`" or "`1`", depending on the state of the token. Similarly, a token with a floating-point native format would result in a string containing the string representation of the floating-point value.
As a last example, inquiring WING token `CH_1_NAME` will return the string currently used for naming channel 1. Please note that the `str` variable should provide enough space to collect the data returned by `wGetTokenString()`.
The function returns `WSUCCESS` if the requested operation was successful, other values can be returned, such as `WZERO` if no suitable format was found for adapting token value to `str`, or `WSEND_TCP_ERROR` if an error took place while communicating with WING. Attempting to get a value from a token of type `NODE` will return `WNODE`.

*int wGetTokenDef(wtoken token, int *num, unsigned char* str)*

The `wGetTokenDef()` function interrogates WING token `token` to get its currently associated definition in raw form. `wGetTokenDef()` will block for 200 milliseconds or until a token `token` is encountered in the receiving queue. The received token definition follows the description presented earlier in this document[84].
The returned data consists of the number of bytes `num` contained in an array of bytes `str`. Note that `str` is allocated by the `wGetTokenDef()` function; it is therefore the responsibility of the calling application to free the allocated memory when no longer needed. Parsing this data is left to the application, and follows the description for node definition response.
The function returns `WSUCCESS` if the requested operation was successful; `wGetTokenDef()` will return `WMEMORY` if it cannot allocate memory for the data to be returned, or `WRECV_ERROR` if the data cannot be recovered; Would `wGetTokenDef()` return `WRECV_ERROR` when you expect data being available, it may be worth attempting a second call to the function as a timeout may have occurred with the first/previous call.

[84] See "Node Definition Response" in the "WING native / binary data interface" chapter

The `Get()` functions presented above are all "one shot read" functions so to speak; They request data from WING, and wait for the right token to appear in the receiving queue. They will return the buffer content, adapting it to the requested type of data when applicable. This is a simple way to gather information from the console, but comes with a caveat if someone is also manipulating (locally or remotely) the desk. Indeed, as other changes take place and assuming your communication channel is in an 'open' state (i.e., your last communication with WING is less than 10s old), the console will natively send you changes that are taking place, resulting of the local or remote changes operated onto the desk.
So, when a "one shot read" request arrives and is served, it will sort through the received data for the expected token, and in doing this will discard the data received prior to finding the correct token.

wapi therefore provides another set of `Get()` functions for applications requiring a finer time control over the data they exchange with WING. In this new set of functions, the `Get()` instance will, as for the non-timed versions, gather information from WING and filter the possibly multiple[85] received tokens for the first one matching the specified token provided at call time, for a given amount of time only. Only when the specified token is received or time has expired (whichever comes first) will the function process the data it received, if available.

*int wGetTokenTimed(wtoken token, wtype *type, wvalue *value, int timeout)*

`wGetTokenTimed()` is equivalent to its blocking sibling function call `wGetToken()`, but will block only for up to `timeout` microseconds; If no data corresponding to token `token` is received during that amount of time, the function will return `WZERO`. If a token `token` is received withing the time allocated by `timeout`, the function will parse data and return it as in the case of `wGetToken()`.

*int wGetTokenFloatTimed(wtoken token, float *fval, int timeout)*

The `wGetFloatTimed()` function is similar to the `wGetTokenFloat()` function in the sense it aims at retrieving from WING data and adapt it to floating-point format before returning it to `fval`;
But it will do so over a period `timeout`, expressed in µs (microseconds).
As long as timeout is not reached, the function is inquiring WING for data; If after timeout has expired, no data appears available, a value of `WZERO` is returned.
If on the contrary data is available from WING, the function will check if the data token is the correct one; it will treat the data as done in the `wGetTokenFloat()` function; i.e. the value retrieved from WING is converted to float format as expected by the `fval` variable.
For example, inquiring WING token `CH_1_PEQ_1F` will return the current value of the token as a float value in `fval`. Inquiring WING token `CH_1_PEQ_ON` will result in a value of `0.0` or `1.0`, depending on the state of the token.
On the other hand, inquiring WING token `CH_1_NAME` will most likely return a value of `0.0` and a status of `WZERO`; In some cases (i.e., you set the name to be string **"123.7"** for example) you may get a valid floating-point value returned.
The function returns `WSUCCESS` if the requested operation was successful, other values can be returned, such as `WZERO` if no suitable format was found for adapting token value to `fval`, or `WSEND_TCP_ERROR` if an error took place while communicating with WING. Attempting to get a value from a token of type `NODE` will return `WNODE`.

[85] Can literally be hundreds

If data is available from WING and the data token is not the expected one, the function discards data and inquires WING for new data. The above takes place as long as timeout is not reached.

*int wGetTokenIntTimed(wtoken token, int *ival, int timeout)*

The `wGetTokenIntTimed()` function is similar to the `wGetTokenInt()` function in the sense it aims at retrieving from WING data and adapt it to floating-point format before returning it to `ival`;
But it will do so over a period `timeout`, expressed in µs (microseconds).
As long as timeout is not reached, the function is inquiring WING for data, if no data appears available, a value of `WZERO` is returned.
If on the contrary data is available from WING, the function will check if the data token is the correct one; it will treat the data as done in the `wGetTokenInt()` function; i.e. the value retrieved from WING is adapted to float format as expected by the `ival` variable.
For example, inquiring WING token `CH_1_PEQ_1F` will return the current value of the token as an int value in `ival`. Inquiring WING token `CH_1_PEQ_ON` will result in a value of `0` or `1`, depending on the state of the token. On the other hand, inquiring WING token `CH_1_NAME` will return a value of `0` and a status of `WZERO`; In some cases (i.e., you set the name to be string **"12"** for example) you may get a valid int value returned.
The function returns `WSUCCESS` if the requested operation was successful, other values can be returned, such as `WZERO` if no suitable format was found for converting the retrieved value to `ival`, or `WSEND_TCP_ERROR` if an error took place while communicating with WING. Attempting to get a value from a token of type `NODE` will return `WNODE`.
If data is available from WING and the data token is not the expected one, the function discards data and inquires WING for new data. The above takes place as long as timeout is not reached.

int wGetTokenStringTimed(wtoken token, char str, int timeout)*

The `wGetTokenStringTimed()` function is similar to the `wGetTokenString()` function in the sense it aims at retrieving from WING data and adapt it to string format before returning it to `str`;
But it will do so over a period `timeout`, expressed in µs (microseconds).
The function is inquiring WING for data until a timeout `timeout` is reached, if no data appears available, a value of `WZERO` is returned.
If on the contrary data is available from WING, the function will check if the data token is the correct one; `wGetTokenStringTimed()` will then treat the data as done in the `wGetTokenString()` function; The value retrieved from WING is adapted to string format as expected by the `str` variable. Similar restrictions and conversion rules apply. For example, inquiring WING token `CH_1_PEQ_1F` will return the current value of the token as a string in `str`. Inquiring WING token `CH_1_PEQ_ON` will result in a 1-character string of "`0`" or "`1`", depending on the state of the token. Similarly, a token with a floating-point native format would result in a string containing the string representation of the floating-point value. As a last example, inquiring WING token `CH_1_NAME` will return the string currently used for naming channel 1. Attempting to get a value from a token of type `NODE` will return `WNODE`.
If data is available from WING and the data token is not the expected one, the function discards data and inquires WING for new data. The above takes place as long as timeout is not reached.

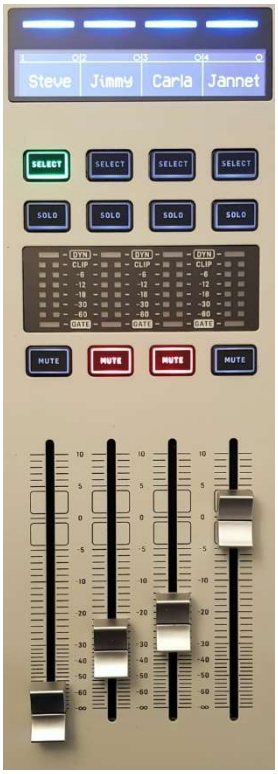
A Small Program Example

Let's program! Assume you need to programmatically change the name of channels and mute/unmute the respective channels from data contained in a file. Let's consider the file also contains initial channel faders, and covers channels 1 to 4. The file can be a text file such as:

```
Steve  0  -144.0
Jimmy  1  -30.0
Carla  1  -22.0
Jannet 0  -100.0
```

This is C source code; easy to understand and translate if needed to other programming languages.
We show on the right of the page the resulting channel strips 1-4:

```
#include <stdio.h>
#include <string.h>
//
#include "wapi.h"
#include "wext.h"
//
int main() {
    wtoken ntoken[] = {CH_1_NAME, CH_2_NAME, CH_3_NAME, CH_4_NAME};
    wtoken mtoken[] = {CH_1_MUTE, CH_2_MUTE, CH_3_MUTE, CH_4_MUTE};
    wtoken ftoken[] = {CH_1_FDR, CH_2_FDR, CH_3_FDR, CH_4_FDR};
    char   wingip[24] = "";    int    mute;
    float  fader;
    char   name[24];
    FILE*  fd;
    //
    // we don't know the IP of our console…
    if (wOpen(wingip)!= WSUCCESS) exit(1);
    printf("WING found at IP: %s\n", wingip);
    // open the file for reading
    if ((fd = fopen("file", "r")) != 0) {
        for (int i = 0; i< 4; i++) {
            // get data from the file
            fscanf(fd, "%23s %i %f", name, &mute, &fader);
            printf("%s %i %f\n", name, mute, fader);
            // set/send values to WING;
            // we don't care about the returned status
            wSetTokenString(ntoken[i], name);
            wSetTokenInt(mtoken[i], mute);
            wSetTokenFloat(ftoken[i], fader);
        }
    }
    fclose(fd);
    wClose();
    exit(0);
}
```



Event-driven updates

There are times and situations when WING will send data to your program. This has been explained above: As soon as you are connected to WING and have exchanged data with it, the connection will stay in an open state for 10s, unless you specifically establish and close the TCP connection around your work. While this will help, it will not prevent WING to send you data while the TCP link is active, and is certainly not an effective way to manage data as you will send more resources in opening/closing the connection than in time sending or receiving data.
wapi provides additional API functions to manage event driven applications. These are managed around the notion of 'main loop' as often found in IOT devices running Arduino devices, in standard Linux or Windows applications where a main loop ensures the management of all events coming from devices connected to your computer (mouse, keyboard, etc.). WING data can be treated just as any other event.
API calls are therefore available to keep a connection between your application and WING alive, as well as to get data from WING, effectively emptying the event queue of the communication with the console. This will be assured with the `wKeepAlive()` and the `wGetVoidPTokenTimed()` function calls presented below.

int wKeepAlive

`wKeepAlive()` maintains the connection between WING and the calling program so data issued by the console with no request initiated by the program can be received in a main loop, or over an extended period of time beyond 10s[86].
In fact, this function can be called as often as you like and will optionally performs a small exchange with the console, based on an internal timer. The elapsed time between two effective exchanges of data with the console depend on the value of `wKeepAlive_TIMEOUT` which is part of the `wapi.h` file.
The `wKeepAlive()` function returns `WSUCCESS` if a valid exchange took place to renew a 10 seconds working communication, or `WZERO` if no exchange was necessary. The function can also return the values of `WSEND_ERROR` or `WRECV_ERROR` if communication was not successful.

*int wGetParsedEvents(wTV *tv, int maxevents)*

The `wGetParsedEvents()` API call is a specific `Get` function. Unlike other `Get` functions previously presented in this document, it does not expect data from a specific token, nor a specified format in which the data from the console should be converted to. The function will check the WING receive event queue for data and will only return when data is received by removing the events available from the oldest queue record. If no valid data is found, the function eventually returns with a network error.
When data is available in the event queue events are retrieved in a FIFO order, and the token, type and data associated to events are returned to the calling application using the `tv` structure array. The function returns the number of events parsed if data has been returned to the calling program, `WZERO` if no events were found. It can also return `WMEMORY` on memory allocation errors or `WRECV_ERROR` on TCP read errors. The parameter `maxevents` represents the maximum number of entries `tv` can accept; The function will allocate memory for its event read buffer to match that size.

[86] The actual value in the wapi library may vary (but less than 10s) to ensure stability in event driven communications

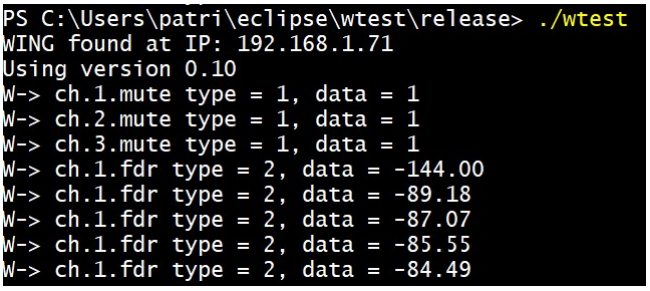
*int wGetParsedEventsTimed(wTV *tv, int maxevents, int timeout)*

The `wGetParsedEventsTimed()` API call is similar to the `wGetParsedEvents()` function, but will returned after a maximum time of `timeout` in µs. Like in the case of the `wGetParsedEvents()` function, `wGetParsedEventsTimed`() does not expect data from a specific token, nor a specified format in which the data from the console should be converted to. The function will check the WING receive event queue for data and will only return when data is received by removing the events available from the oldest queue record. If no data is found before a timeout of `timeout` µs, the function returns with a value `WZERO`.
If data is available in the event queue, events are retrieved in a FIFO order, and the token, type and data associated to events are returned to the calling application using the `tv` structure array. The function returns the number of events parsed if data has been returned to the calling program, `WZERO` if no events were found. It can also return `WMEMORY` on memory allocation errors or `WRECV_ERROR` on TCP read errors. The parameter `maxevents` represents the maximum number of entries `tv` can accept; The function will allocate memory for its event read buffer to match that size.

In a typical, simple example of use of the two API calls shown in the following paragraph, the main loop is replaced with a `while(1){}` statement.

```
#include <stdio.h>
#include <string.h>
//
#include "wapi.h"
#include "wext.h"
//
int main() {
    int   i, j;
    char  wingip[24] = "";
    wTV   TV[100];

    if ((i = wOpen(wingip)) != WSUCCESS) return(-1);
    printf("WING found at IP: %s\n", wingip);
    printf("Using version %i.%i\n", wVer()/256, wVer()&15);

    while (1) {
        wKeepAlive();
        //
        if ((i = wGetParsedEventsTimed(TV, 100, 1000)) > 0) {
             for (j = 0; j < i; j++) {
                printf("W-> %s type = %i, data = ", wGetName(TV[j].token), TV[j].type);
                if (TV[j].type == I32) printf("%i\n", TV[j].d.idata);
                if (TV[j].type == F32) printf("%.2f\n", TV[j].d.fdata);
                if (TV[j].type == S32) {
                    if (TV[j].d.sdata) {
                       printf("%s\n", TV[j].d.sdata);
                       free(TV[j].d.sdata);
                    }
                }
                fflush(stdout);
             }
        } else {
            if (i != WZERO) {
                printf("Error = %i\n", i); fflush(stdout);
            }
        }
    }
    return 0;
}
```

An example of (partial) output of the code snippet above, after launching the program and manipulating `ch.1`, `2` and `3` mutes and `ch. 1` fader:


PS C:\Users\patri\eclipse\wtest\release> ./wtest
WING found at IP: 192.168.1.71
Using version 0.10
W-> ch.1.mute type = 1, data = 1
W-> ch.2.mute type = 1, data = 1
W-> ch.3.mute type = 1, data = 1
W-> ch.1.fdr type = 2, data = -144.00
W-> ch.1.fdr type = 2, data = -89.18
W-> ch.1.fdr type = 2, data = -87.07
W-> ch.1.fdr type = 2, data = -85.55
W-> ch.1.fdr type = 2, data = -84.49

Nodes

In many applications as well as in browsing over the `JSON` data structure, one can easily envision it would be interesting for optimization purposes to get and set a group of attributes at once, rather than establishing communication requests for each single parameter.

Nodes were introduced in the X32 family to enable this functionality, and have been widely used in several applications for controlling the desk; In the case of WING this may be even more interesting due to the very large number/volume of parameter data available as one unrolls each branch in the `JSON` tree opening a new level of nodes and parameters. Each branch of the `JSON` tree can be walked through by a program, resulting in a (sometimes very large) set of {`token`, `value`} sets and a way to represent the depth in the hierarchical tree the reported sets are issues from.

We show below the node data extracted (using a `wapi` call) for a few nodes:

```
wing_root: {$stat{}, cfg{}, $syscfg{}, io{}, ch{}, aux{}, bus{}, main{}, mtx{}, dca{}, fx{},
cards{}, play{}, rec{}, $ctl{}, $globals{}}

node $stat, size: 212,
.A.stat=-,dev=,.B.stat=-,dev=,.C.stat=-,dev=,.lock=1,ppm=0,solo=0,sip=0,rtcerr=0,time='15:41:28',d
ate=2025-02-13,usbstate=IDLE,usbvolname=KINGSTON,sc_stat=-,sc_devices=,sc_upcnt=32,sc_dncnt=32,sc_
uprout=,rmt_a=,rmt_b=,rmt_c=,~~~~

node $syscfg, size: 208,
.consolename=HMS-01,logflags=toto,ipmode=DHCP,ip0=169,ip1=254,ip2=24,ip3=25,msk0=255,msk1=255,msk2
=0,msk3=0,gw0=192,gw1=168,gw2=0,gw3=254,tcplock=0,usbh_spd=HS,eth_cfg=SEPARATED,opt_mod=DANTE,~
```

As mentioned above, some nodes such as `ch`, are very large (more than 80k characters).

The set of values in a node list can be variable depending on the options (effects for example) loaded in the console at the time of the call, but all tokens are fixed and only contain known data types; A node can be set and retrieved as a single line of text with pre-formatted data, making it easy to store and manage in applications. `wapi` offers two methods of saving node data sets from the console, the first one is returning data like what is obtained using `OSC` (text data); the other one is more suitable to direct use from a compiled program, with binary data saved in specific structures containing the token and its respective data. The first method typically takes 2 or 3 seconds to get all WING data as multiple strings of node data. The second method is probably more suitable for use with `wapi` and is also faster (1 to 2 seconds) as no formatting is involved in saving the data returned by the console.

The following functions list API entries to use WING nodes as defined above.

*int wSetNode(char *str)*

The `wSetNode()` function parses the string contained in `str` according to the format used in OSC nodes; For example, a string such as `/ch.1.fdr=8.5,mute=1,/bus.1.fdr=5.0,.2.fdr=0.5` will set fader of channel 1 to the 8.5dB value and mute the channel. Bus 1 fader will be set to 5dB and bus 2 fader will be set to 0.5dB.
Each `parameter=value` group is separated by a '`,`' character, the '/' character represents the root of the `JSON` tree, and '`.`' characters are used to navigate up and down within the `JSON` tree. String type values containing space characters should be encapsulated within '"' characters, such as in `/ch.1.name="space name"`
The function returns a status `WSUCCESS` if the string was processed with no errors; It will return `WNODE` if a token or value provided with the string `str` is not valid. The function can also report other errors if communication issues were detected. `str` must be `\0` ended. Please see a code example using `wSetNode()` further in this document.

*int wSetNodeFtomTVArray(wTV *array, int nTV)*

The `wSetNodeFromTVArray()` function sends updates to `WING` in a single network exchange from the `nTV` elements in `wTV` (see below) array `array`; This is a great way to improve network performance. Although the function is the symmetrical to `wGetNodeToTVArray()`, it can accept hierarchically organized elements or uncorrelated elements as long as they are not nodes and contain valid tokens-values sets. The function returns `WSUCCESS` or an error if one takes place during allocating, preparing, or sending the resulting network buffer to `WING`.

*int wSetBinaryNode (unsigned char *array, int len)*

The `wSetBinaryNode()` function will load from `array` the `len` bytes of binary data commands to be executed by the desk in a single call. The function returns the number of bytes sent to the console on success or an error if one takes place during sending the buffer to `WING`.
The length value returned by the call can be greater than the value of `len`. This results from the function call possibly adding escape characters specific to Wing native/binary protocol.

A typical use for this call is external scene management[87] such as in the code snippet below.

```
void WRestoreS() {
    unsigned char node[256000];
    int i, j;
    //
    if ((fd = fopen("Scene.scn", "rb")) != NULL) {
        fread(&i, sizeof(int), 1, fd);
        fread(node, i * sizeof(unsigned char), 1, fd);
        if ((j = wSetBinaryNode(node, i)) < i) {
            printf("Error: %d only bytes sent vs. %d\n", j, i);
        } else {
            printf("Restored Scene\n");
        }
        fclose(fd);
    } else {
        printf("Cannot open Scene.scn\n");
    }
    Return;
}
```

[87] As opposed to WING internal Show files and Scene entities (see dedicated chapter)

*int wGetNode(wtoken node, char *str)*

The `wGetNode()` function will return in `str` a string of values separated formatted as in the `OSC` node convention and corresponding to the node token `node`.
`str` must be large enough to accept the characters returned by the call. The function returns a status `WSUCCESS` if the node was processed with no errors; It will return `WTOKEN` if the token provided is not a valid node and `WNODE` if an error occurs during parsing the data received from the console. The function can also report other errors if communication issues were detected. The line of text returned by the function end with a line-feed and a `\0` byte.
Note that it may not be possible to directly send node data as a string received with a `wGetNode()` using `wSetNode()`; Indeed, some nodes have variable/dynamically assigned parameters, such as in the case of equalizer models, and the parameter names reported by `wGetNode()` must be changed to a numerical list of parameters prior to being passed as a parameter string to `wSetNode()`; See "*Dynamic parameters anonymization in wapi*" further in this document.

*int wGetNodeToTVArray (wtoken node, wTV *array)*

The `wGetNodeToTVArray()` function will return in TV, an array of structures `wTV` (see below), all values respective of their corresponding token and part of the node token `node`.
`array` must be large enough to accept the data returned by the call (see below for the number of elements for each level-1 node). The function the number of tokens in the array `array` if the node was processed with no errors; It will return `WTOKEN` if the token provided is not a valid node and `WNODE` if an error occurs during parsing the data received from the console. The function can also report other errors if communication issues were detected.

Below is an *indicative* value of the number of `wTV` structures in the returned arrays for each level-1 node of the console at the time of this writing; The sum of the values following 'size:' give an idea of the number of parameters the console knows.

```
node $stat, size: 23
node cfg, size: 194
node $syscfg, size: 25
node io, size: 6054
node ch, size: 12400
node aux, size: 2008
node bus, size: 2736
node main, size: 440
node mtx, size: 728
node dca, size: 144
node mgrp, size: 16
node fx, size: 624
node cards, size: 74
node play, size: 18
node rec, size: 7
node $ctl, size: 5328
node $globals, size: 11
```

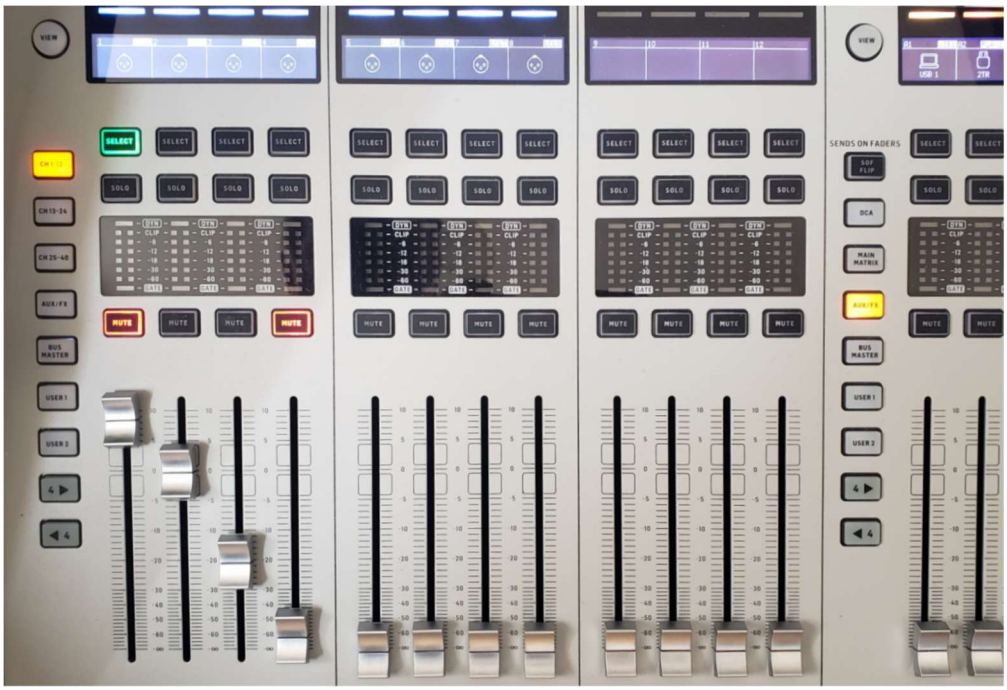
Below is a C code source example of use of `WSetNode()`; The program will set the faders of channels 1 to 4 to different positions, unmute channels 1 and 3, channel 2 mute is unchanged and channel 4 will be muted. DCA 1 is muted and its fader is set to 1dB.

```
/*
 * wtest.c
 *
 *  Created on: Oct. 18, 2020
 *      Author: Patrick-Gilles Maillot
 */
//
#include <stdio.h>
#include <string.h>
//
#include "wapi.h"
#include "wext.h"
//
int main() {
    int    i;
    char      wingip[24] = "";
    char      wtest1[64] = "/ch.1.fdr=8.5,mute=1,name=toto,.2.fdr=0,/dca.1.fdr=1.0,mute=0";
    char      wtest2[64] = "/ch.3.fdr=-20,mute=0,name=,.4.fdr=-60,mute=1";
    //
    if ((i = wOpen(wingip)) != WSUCCESS) exit(1);
    printf("WING found at IP: %s\n", wingip);
    unsigned int ui = wVer();
    printf("Using wapi ver: %i.%i.%i-%i\n", ui >> 24, (ui & 0xff0000) >> 16,
                                            (ui & 0xff00) >> 8, ui & 0xff);
    //
    i = wSetNode(wtest1);
    printf("result = %d, initial data: %s\n", i, wtest1);
    i = wSetNode(wtest2);
    printf("result = %d, initial data: %s\n", i, wtest2);
    return(0);
}
```

A listing of the program when ran:

```
PS C:\Users\patri\eclipse\wtest\release> ./wtest
WING found at IP: 192.168.1.90
Using wapi ver: 3.0.6-4
result = 1, initial data: /ch.1.fdr=8.5,mute=1,name=toto,.2.fdr=0,/dca.1.fdr=1.0,mute=0
result = 1, initial data: /ch.3.fdr=-20,mute=0,name=,.4.fdr=-60,mute=1
PS C:\Users\patri\eclipse\wtest\release>
```

Below the resulting state of the console, starting from an init state:



Requesting the full set of nodes from a freshly initialized console[88] results in a file of 200000+ characters, and is therefore a lot of data to manage. Over WIFI, it takes about 2 seconds to execute a full dump as OSC-like node data and 1 to 2 seconds to retrieve a full dump as `wTV` structures.

We show below a typical example of the `OSC`-like node string for `ch.1` returned by **wapi** when using `wGetNode`():

```
node ch:
.1.in.set.$mode=M,srcauto=0,altsrc=0,inv=0,trim=0.00,bal=0.00,$g=0.00,$vph=0,dly=0.00,.conn.grp=LC
L,in=1,altgrp=OFF,altin=1,..flt.lc=0,lcf=100.24,hc=0,hcf=10018.26,tf=0,mdl=TILT,tilt=0.00,.clink=0
,col=1,name="",icon=1,led=1,mute=0,fdr=144.00,pan=0.00,wid=100.00,$solo=0,$sololed=0,solosafe=0,mo
n=A,proc=GEDI,ptap=4,$presolo=0,peq.on=0,1g=0.00,1f=99.69,1q=2.00,2g=0.00,2f=999.25,2q=2.00,3g=0.0
0,3f=10016.53,3q=2.00,.gate.on=0,mdl=GATE,thr=40.00,range=40.00,att=10.00,hld=10.00,rel=199.40,acc
=0.00,ratio=1:3,.gatesc.type=OFF,f=1002.37,q=2.00,src=SELF,tap=IN,$solo=0,.eq.on=0,mdl=STD,mix=100
.00,$solo=0,$solobd=1,lg=0.00,lf=80.20,lq=2.00,leq=SHV,1g=0.00,1f=200.00,1q=2.00,2g=0.00,2f=601.39
,2q=2.00,3g=0.00,3f=1499.79,3q=2.00,4g=0.00,4f=3990.52,4q=2.00,hg=0.00,hf=11994.42,hq=2.00,heq=SHV
,.dyn.on=0,mdl=COMP,mix=100.00,gain=0.00,thr=10.00,ratio=3.00,knee=3,det=RMS,att=50.00,hld=20.00,r
el=152.57,env=LOG,auto=1,.dynxo.depth=6.00,type=OFF,f=1002.37,$solo=0,.dynsc.type=OFF,f=1002.37,q=
2.00,src=SELF,tap=IN,$solo=0,.preins.on=0,ins=NONE,$stat=,.main.1.on=1,lvl=0.00,.2.on=0,lvl=0.00,.
3.on=0,lvl=0.00,.4.on=0,lvl=0.00,..send.1.on=0,lvl=144.00,pon=0,ind=0,mode=PRE,plink=0,pan=0.00,wi
d=100.00,.2.on=0,lvl=144.00,pon=0,ind=0,mode=PRE,plink=0,pan=0.00,wid=100.00,.3.on=0,lvl=144.00,po
n=0,ind=0,mode=PRE,plink=0,pan=0.00,wid=100.00,.4.on=0,lvl=144.00,pon=0,ind=0,mode=PRE,plink=0,pan
=0.00,wid=100.00,.5.on=0,lvl=144.00,pon=0,ind=0,mode=PRE,plink=0,pan=0.00,wid=100.00,.6.on=0,lvl=1
44.00,pon=0,ind=0,mode=PRE,plink=0,pan=0.00,wid=100.00,.7.on=0,lvl=144.00,pon=0,ind=0,mode=PRE,pli
nk=0,pan=0.00,wid=100.00,.8.on=0,lvl=144.00,pon=0,ind=0,mode=PRE,plink=0,pan=0.00,wid=100.00,.9.on
=0,lvl=144.00,pon=0,ind=0,mode=PRE,plink=0,pan=0.00,wid=100.00,.10.on=0,lvl=144.00,pon=0,ind=0,mod
e=PRE,plink=0,pan=0.00,wid=100.00,.11.on=0,lvl=144.00,pon=0,ind=0,mode=PRE,plink=0,pan=0.00,wid=10
0.00,.12.on=0,lvl=144.00,pon=0,ind=0,mode=PRE,plink=0,pan=0.00,wid=100.00,.13.on=0,lvl=144.00,pon=
0,ind=0,mode=PRE,plink=0,pan=0.00,wid=100.00,.14.on=0,lvl=144.00,pon=0,ind=0,mode=PRE,plink=0,pan=
0.00,wid=100.00,.15.on=0,lvl=144.00,pon=0,ind=0,mode=PRE,plink=0,pan=0.00,wid=100.00,.16.on=0,lvl=
144.00,pon=0,ind=0,mode=PRE,plink=0,pan=0.00,wid=100.00,..postins.on=0,mode=FX,ins=NONE,w=0.00,$st
at=,.tags="",$fdr=144.00,$mute=0,$muteovr=0,
```

[88] FW 1.10

*int wGetBinaryNode (wtoken node, unsigned char *array, int maxlen)*

The `wGetBinaryNode()` function will return in `array` the raw, binary data corresponding to the `WING` node selected with token `node`. The storage buffer `array` must be large enough to accept the data returned by the call, up to `maxlen` bytes. The function returns the number of bytes saved in `array`, or an error status if less or equal to 0.
Note that token `node` can represent a node or a parameter.
A typical use for this call is external scene management[89] such as in the code snippet below.

```
void WSaveS() {
    unsigned char node[2048];
        int i;
    //
    if ((fd = fopen(Scene.scn, "wb")) != NULL) {
        // Get the node data for CH_1, composed of many 0xd7<Wing data> parts
        if ((i = wGetBinaryNode(CH_1, node, 2048)) < WSUCCESS) {
            fclose(fd);
            printf("Error reading node %d\n", tarray[ta]);
            return;
        }
        fwrite(&i, sizeof(int), 1, fd);
        fwrite(node, i * sizeof(unsigned char), 1, fd);
        printf("Saved scene\n");
        fclose(fd);
    } else {
        printf("Cannot create Scene.scn\n");
    }
    return;
}
```

*int wGetBinaryData (char *str, unsigned char *array, int maxlen)*

The `wGetBinaryData()` function will return in `array` the raw, binary data corresponding to the `WING` node or parameter represented by its text description `str`. The storage buffer `array` must be large enough to accept the data returned by the call, up to `maxlen` bytes.
`str` is a string of characters enabling to select a single WING node or parameter, for example:

"/ch" (a node),
"/ch.1.eq" (a node), or
"/ch.1.name" (a parameter).

The function returns the number of bytes saved in `array`, or an error status if less or equal to 0.

[89] As opposed to WING internal Show files and Scene entities (see dedicated chapter)

Meters

WING offers many measurement points along the digital audio path; As a result, there are numerous meters. As briefly presented in a table earlier in this document, every Channels, Aux, Bus, Main and Matrix strip offers no less than 8 meters: input left & right, output left & right, gate key & gain, and dyn key & gain. This alone represents 608 meters that can be retrieved, and there are even more with V2 (version 2) meters, and many more meter data from other metering points.
The network data path for getting meter values is separated from the main network communication to keep things simpler for the programmer and sound engineer.
Meter data is transmitted to a `UDP` port chosen by the user. When selecting which meters to receive, the user associates an ID to the request, enabling simpler identification of the received data. As soon as a valid meter request is received, WING will send back the respective meter data for 5 seconds, every approximately every 50ms. To continue or continuously receive a set of meter data, the user must renew the request for data by issuing a simple renew command containing the ID of the requested meter set.

Meters API

wapi offers a small set of function calls to help programmers manage meter data. it hides the networking complexity and proposes a simple way of selecting what meter to get back from the digital console. Meter data will be provided back to the application in the form of a buffer of values, decoding data being left to the application.

int wMeterUDPPort (int wport)

The `wMeterUDPPort()` API call enable users to select the UDP port WING will send meter data to. It also prepares the `wapi` internal network for receiving meter data and being able to return data to the user application. `wport` is a standard UDP port and must be available for receiving data. The function returns WSUCCESS if everything is set correctly or will return an error value if the request was not successful.

*int wSetMetersRequest(int reqID, unsigned char *wMid)*

`wSetMetersRequest()` must be called in order to start receiving meter data. The API associates a request ID `reqID` to a selection of meters to receive. The request ID helps renewing the request for data and sorting through potentially multiple data sets sent by the console. The `wMid` parameter holds the selection of meters that can be recovered from WING in an array of `30` bytes. Each bit (from left to right) in the array of `29` bytes represents a meter family that can be received from the console, and is shown in the table below:

byte index	bits	selection
0-4	1-40	Channel 1-40
5	1-8	Aux 1-8
6-7	1-16	Bus 1-16
8	1-4	Main 1-4
9	1-8	Matrix 1-8
10-11	1-16	DCA 1-16
12-13	1-16	FX proc 1-16
14-15	1-16	Source input 1-16
16-17	1-11	Output 1-11
18	1	Monitor
19	1	RTA
20-24	1-40	Channel V2 1-40
25	1-8	Aux V2 1-8
26-27	1-16	Bus V2 1-16

28	1-4	Main V2 1-4
29	1-8	Matrix V2 1-8

For example, a C source language array declaration as follows will request meters for channels 1 and 40:

```
unsigned char   mbits[30] = {0x80, 0, 0, 0, 0x01, 0};    // bytes indexes 5 to 29 are 0
```

int wRenewMeters(int reqID)

The `wRenewMeters()` API call is used to renew a previous request for meter data; This function should be called every 5 seconds maximum in order to avoid losing meter data if continuous receiving is expected. The `reqID` parameter must be previously defined with a call to `wSetMetersRequest()`. The function returns `WSUCCESS` if the request is accepted, or will return other error status values otherwise.

*int wGetMeters(unsigned char *buf, int maxlen, int timeout)*

`wGetMeters()` will check if meter data has been received or is available. The call can be blocking or un-blocking depending on the value of `timeout`. A `timeout` of `0` will block the application in reading mode until data is available. A non-zero value of `timeout`, expressed in micro-seconds will return after the provided value and return to the caller with a value of `WZERO (0)` if no data is available or will return sooner with the actual number of bytes read available in `buf.`
The `maxlen` parameter indicates the maximum number of bytes `buf` can hold. It is the responsibility of the application to ensure `buf` is large enough to accept `maxlen` bytes.

The data returned by the wGetMeters() function is coded as follows:

```
<reqID><[meter data group][meter data group]  … >
```

Each `meter data group` is composed of several big-endian 16bits integers typically representing meter values expressed in `1/256`th of a dB, or otherwise returned data (for ex. `Gate led` returns `0` or `1`).

The table below provides the number and origin of each meter data for each of the possible meter groups:

Group name	Contents
channel aux bus main matrix	input left input right output left output right gate key gate gain dyn key dyn gain
dca	pre fader left pre fader right post fader left post fader right
fx	input left input right output left output right state meters (1…6)
source	source group levels (i.e. local ins: 8 meters)
output	output group levels (i.e. local outs: 8 meters)
monitor	solo bus left solo bus right

	mon 1 left mon 1 right mon 2 left mon 2 right
rta	rta slot meters (120)
channel V2 aux V2 bus V2 main V2 matrix V2	input left input right output left output right gate key gate gain gate led dyn key dyn gain dyn state automix gain

Below is an example of buffers received after requesting meter data for channel 1 and using different sources, with `Ch 1` fader set to `+3dB`.
As received data uses signed `16bits` (`[-32768…+32767]`) and is expressed in `1/256`th of dB, the actual meter value can be calculated as `<int16>/256`.
Note that fx return a different format for their meters, with `value` in dB `= return value * 6.0 / 2048`.

```
                                    gate gate  dyn  dyn
          <reqID>  inL  inR outL outR  key gain  key gain

W→ 20 B: 00000002 9bb2 9bb2 8000 8000 9b7c 0000 8000 0000 (no input)
                  -100 -100 -128 -128 -100    0 -128    0
W→ 20 B: 00000002 f9fb f9fb fcfb fcfb f9fb 0000 ee01 0000 (OSC 1kHz, -6dB)
                    -6   -6   -3   -3   -6    0  -17    0
W→ 20 B: 00000002 d7fd d7fd dafd dafd d7fd 0000 aa02 0000 (OSC 1kHz, -40dB)
                   -40  -40  -37  -37  -40    0  -85    0
```

RTA test program

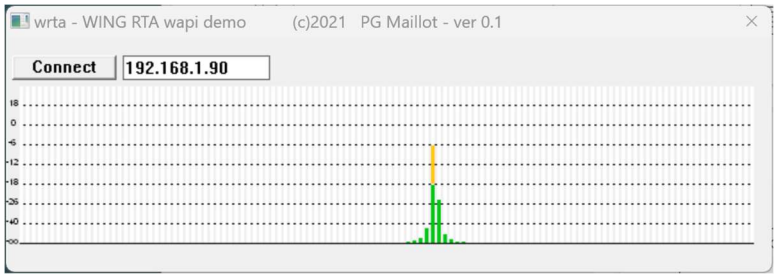
We show here is a small C / Windows program example showing how to get and display RTA. The scaling of meter data is tweaked in order to provide better readability, but isn't meant to be dB accurate.

```
/*
 * wrta.c
 *
 *  Created on: May 9, 2020
 *      Author: Patrick-Gilles Maillot
 *
 *
 * History
 *  ver 0.0: initial release
 *  ver 0.1: changed color scheme and used a pow() function to better match WING
 *             results out of raw meter data;
 *           updated to meters V2
 */
#include <windows.h>
#include <stdio.h>
#include <sys/time.h>
#include <math.h>
#include "../wapi/wapi.h"
#include "../wapi/wext.h"
//
// Windows Declarations
WINBASEAPI HWND WINAPI GetConsoleWindow(VOID);
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
//
HINSTANCE       hInstance = 0;
HWND            hwndipaddr, hwndconx;
HDC             hdc;
PAINTSTRUCT     ps;
MSG             wMsg;
HFONT           hfont;
HPEN            wnpen;                  // no line
HBRUSH          gBrush, rBrush, yBrush, wBrush;    // Green, red, yellow, white
int             keep_running = 1;       // mainloop control
int             ready = 0;              // Ready flag after connect OK
char            wingip[24] = "";        // Let wapi tell us our IP
int             M_id = 3;               // Meters request ID
int             M_port = 10026;         // Meters UDP port

#define MAXLEN  254                     // enough for RTA (244 bytes)
unsigned char   buf[MAXLEN];            // data buffer
int             len;
//
time_t          before = 0;             // Timers
time_t          now;
//
unsigned char    mbits[30] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                              0, 0, 0, 0, 0, 0, 0, 0, 0, 0x80, // RTA
                              0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
//
// void wRTAMeters()
// A basic RTA, positioned height is 128, width is 5 per freq. (120 Freqs)
// we draw (systematically) rectangles from 0 to 128, varying the color depending on value and
// pre-defined thresholds for yellow and red. We always optionally draw green, yellow, red,
// and finally white
void wRTAMeters(int basex, int basey, int value) {
    int basexx = basex + 5;
    int baseyy = basey + 128;
    basex++;
    // not trying to be accurate, but close to WING data behavior
    //
    if (value < 48) {
        SelectObject(hdc, gBrush);
        SelectObject(hdc, wnpen);
        Rectangle(hdc, basex, baseyy, basexx, baseyy - value);
        SelectObject(hdc, wBrush);
        Rectangle(hdc, basex, baseyy - value, basexx, basey);
    } else if (value < 96) {
```

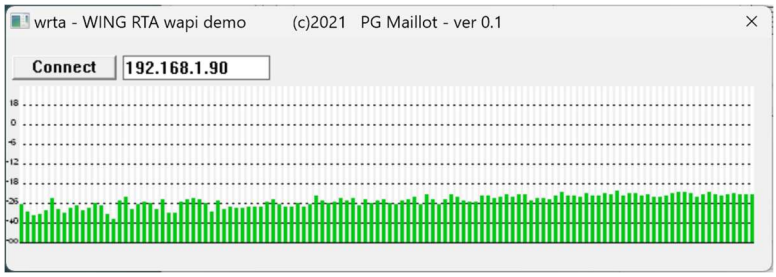
```
        SelectObject(hdc, gBrush);
        SelectObject(hdc, wnpen);
        Rectangle(hdc, basex, baseyy, basexx, baseyy - 48);
        SelectObject(hdc, yBrush);
        Rectangle(hdc, basex, baseyy - 48, basexx, baseyy - value);
        SelectObject(hdc, wBrush);
        Rectangle(hdc, basex, baseyy - value, basexx, basey);
    } else {
        SelectObject(hdc, gBrush);
        SelectObject(hdc, wnpen);
        Rectangle(hdc, basex, baseyy, basexx, baseyy - 48);
        SelectObject(hdc, yBrush);
        Rectangle(hdc, basex, baseyy - 48, basexx, baseyy - 96);
        SelectObject(hdc, rBrush);
        Rectangle(hdc, basex, baseyy - 96, basexx, baseyy - value);
        SelectObject(hdc, wBrush);
        Rectangle(hdc, basex, baseyy - value, basexx, basey);
    }
}
//
// Windows main function and main loop
//
int WINAPI wWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, PWSTR lpCmdLine, int nCmdFile) {
//
    union {
        unsigned char    cc[2];
        short            ii;
    } endian;
    int                  ival = 0;
    float                fval;
    //
    WNDCLASSW wc = {0};
    wc.lpszClassName = L"WING RTA";
    wc.hInstance = hInstance;
    wc.hbrBackground = GetSysColorBrush(COLOR_3DFACE);
    wc.lpfnWndProc = WndProc;
    wc.hCursor = LoadCursor(0, IDC_ARROW);
    //
    RegisterClassW(&wc);
    CreateWindowW(wc.lpszClassName,
        L"wrta - WING RTA wapi demo          (c)2021   PG Maillot - ver 0.1",
        WS_OVERLAPPED | WS_VISIBLE | WS_SYSMENU,
        220, 220, 630, 220, 0, 0, hInstance, 0);
    //
    // Main loop
    while (keep_running) {
        if (PeekMessage(&wMsg, NULL, 0, 0, PM_REMOVE)) {
            TranslateMessage(&wMsg);
            DispatchMessage(&wMsg);
        }
        if (ready) {
            now = time(NULL);               // maintain meters
            if (now > before + 4) {         // by sending
                wRenewMeters(M_id);         // request every less than
                before = now;               // 5 seconds
            }
            // Read meters (if any data) with a timeout of 10ms
            if ((len = wGetMeters(buf, MAXLEN, 10000)) > 0) {
                for (int i = 0; i < 120; i++) {
                    endian.cc[0] = buf[5 + i + i];    // channel in 1
                    endian.cc[1] = buf[4 + i + i];
                    fval = ((float)(endian.ii + 32768)/32768.);
                    ival = pow(fval, 10) * 128;
                    wRTAMeters(10 + i * 5, 40, ival);
                }
            }
        }
    }
    return (int) wMsg.wParam;
}
//
LRESULT CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam) {
    //
```

```
    char *str1[] = {" 18", "  0", " -6", "-12", "-18", "-26", "-40", "-oo"};

    switch (msg) {
    case WM_CREATE:
        hwndconx = CreateWindow("button", "Connect", WS_VISIBLE | WS_CHILD,
                5, 15, 85, 20, hwnd, (HMENU)1, NULL, NULL);
        hwndipaddr = CreateWindow("Edit", NULL, WS_CHILD | WS_VISIBLE | WS_BORDER,
                95, 15, 120, 20, hwnd, (HMENU)0, NULL, NULL);
        break;
    case WM_PAINT:
        hdc = BeginPaint(hwnd, &ps);
        SelectObject(hdc, hfont);
        SetBkMode(hdc, TRANSPARENT);
        for (int i = 0; i < 8; i++) {
            TextOut(hdc, 0, 40 + i*16+10, str1[i], strlen(str1[i]));
            MoveToEx(hdc, 11, 40 + i*16+15, NULL);
            LineTo(hdc, 609, 40 + i*16+15);
        }
        break;
    case WM_COMMAND:
        if (HIWORD(wParam) == BN_CLICKED) {    // user action
            switch (LOWORD(wParam)) {
            case 1:
                if (ready) {
                    keep_running = 0;
                    PostQuitMessage(0);
                } else {
                    // Connect clicked
                    if (wOpen(wingip) != WSUCCESS) exit(1);
                    SetWindowText(hwndipaddr, wingip);
                    // set udp  port to receive UDP data and request meters for channel 1
                    if (wMeterUDPPort(M_port) != WSUCCESS) exit(1);
                    if (wSetMetersRequest(M_id, mbits) != WSUCCESS) exit(1); // Meter req ID 3
                    ready = 1;
                }
                break;
            }
        }
        break;
    case WM_DESTROY:
        keep_running = 0;
        PostQuitMessage(0);
        break;
    }
    return DefWindowProcW(hwnd, msg, wParam, lParam);
}
//
// main program
int main(int argc, char **argv) {
    HINSTANCE hPrevInstance = 0;
    PWSTR pCmdLine = 0;
    int nCmdFile = 0;
    // Hide console window
    ShowWindow(GetConsoleWindow(), SW_HIDE);
    // Set colors
    gBrush = CreateSolidBrush(RGB(0, 200, 20));
    yBrush = CreateSolidBrush(RGB(255, 200, 20));
    rBrush = CreateSolidBrush(RGB(255, 30, 0));
    wBrush = CreateSolidBrush(RGB(255, 255, 255));
    wnpen = CreatePen(PS_NULL, 0, RGB(0, 0, 0));
    hfont = CreateFont(8, 0, 0, 0, FW_MEDIUM, 0, 0, 0,
            DEFAULT_CHARSET, OUT_OUTLINE_PRECIS, CLIP_DEFAULT_PRECIS,
            ANTIALIASED_QUALITY, VARIABLE_PITCH, TEXT("Arial"));
    // Launch program
    wWinMain(hInstance, hPrevInstance, pCmdLine, nCmdFile);
    wClose();
    return 0;
}
```

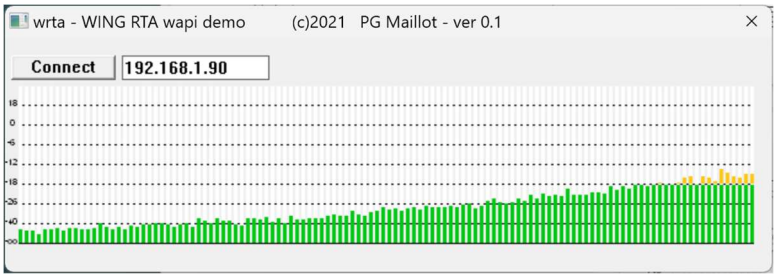
And the resulting displays: Sine wave signal 1kHz:



Pink noise mode:



White noise mode:



Channel strips layers

WING consoles offer a full customization of their control surface or screen, enabling the standard/default settings of course but any strip can be configured to become a different one.

The goal here is not to describe how to use the feature, but to warn the programmer on a specific parameter when using channel strip layers with `wapi`.

The typical set of parameters for Section Left, layer 1 node 1 is the following:

/$ctl/layer/L/1
/$ctl/layer/L/1/ofs
/$ctl/layer/L/1/name
/$ctl/layer/L/1/1
/$ctl/layer/L/1/1/type
/$ctl/layer/L/1/1/i
/$ctl/layer/L/1/1/dst
/$ctl/layer/L/1/1/val

The type can take different values of assigned channel strip type: `CH, BUS, DCA, SEN, FX, MIDI CC`. For all but the type `MIDI CC`, the set of attributes is receiving a fixed name. In the case of `MIDI CC`, the attribute `val` is anonymized as `1`. Therefore, all **wapi** tokens will fit to their counterpart's name, except for `val`, as shown below:

/$ctl/layer/L/1	`$CTL_LAYER_L_1`
/$ctl/layer/L/1/ofs	`$CTL_LAYER_L_1_OFS`
/$ctl/layer/L/1/name	`$CTL_LAYER_L_1_NAME`
/$ctl/layer/L/1/1	`$CTL_LAYER_L_1_1`
/$ctl/layer/L/1/1/type	`$CTL_LAYER_L_1_1_TYPE`
/$ctl/layer/L/1/1/i	`$CTL_LAYER_L_1_1_I`
/$ctl/layer/L/1/1/dst	`$CTL_LAYER_L_1_1_DST`
/$ctl/layer/L/1/1/val	`$CTL_LAYER_L_1_1_1`

Effects and Plugins

WING comes with an impressive number of effects, plugins and emulations that can be used on any channel without costing any FX slots. In every channel, Gate, EQ Compressor can take different processing models you can organize and change on the fly. The following pages below present the different effects and their parameters. For a detailed description of effects and plugins, please refer to the "Processing and Effects Plug-in Guide"[90] on Behringer's website.

Plugins

Plugins entries are directly included with channels, busses, etc. and can either default to WING standard algorithms or adapt to alternative plugins to color your sound or fit your taste when it comes to mixing. Plugins are showing under the main `JSON` structure, only when instantiated. WING **Channel** audio engines enable 4 sorts of plugins: Filter, Gate, EQ and Dynamics. **Bus**, **Main** and **Matrix** audio engines support EQ and Dynamics plugins.

The choice of plugin is represented by the name (or model) of the plugin, as set under the respective "`mdl`" token; After a console reset, the default channel Filter, Gate, EQ and Dynamics plugins will be "`TILT`", "`GATE`", "`STD`", and "`COMP`", respectively, and these can be changed to one of the multiple plugins available within the console (respecting the category they apply to of course).

The choice of plugin is represented by the name (or model) of the plugin, as set under the respective "`mdl`" token; authorized values are:

Filters:

```
TILT EQ, MAXER, AP 90, AP 180
```

Gates/Compressors:

```
WING GATE, WING DUCKER, EVEN 88 GATE, SOUL 9000 GATE, DRAW MORE 241, BDX902 DEESSER,
DYNAMIC EQ, DUAL DYNAMIC EQ, WAVE DESIGNER, SOURCE EXTRACTOR, PSE/LA COMBO, AUTO RIDER,
SOUL WARMTH PRE, WING COMPRESSOR, WING EXPANDER, BDX 160 COMP, BDX 560 EASY, DRAW MORE
COMP, EVEN COMP/LIM, SOUL 9000, SOUL G BUS COMP, RED3 COMPRESSOR, 76 LIMITER AMP, LA
LEVELER, FAIR KID, ETERNAL BLISS, NO-STRESSOR, PIA2250 RACK, LTA100 LEVELER, EVEN 88 COMP,
LMT COMPRESSOR, ONE KNOB COMP
```

Equalizers:

```
WING EQ, SOUL ANALOGUE, EVEN 88 FORMANT, EVEN 84, FORTISSIMO 110, PULSAR, MACH EQ4
```

For a **wapi** program to gain access to plugin parameters, independently from the plugin being installed/loaded at a given slot, the plugin parameter names are being 'anonymized' to names `1…n`, rather than the names that are listed with each single plugin. The actual parameter names for each separate plugin are listed in the plugin description tables later in this document and are preceded with their apparition number in the plugin parameter list; For example, to access the "`range`" value of plugin "`GATE`" used in channel `03`, you would set the token value to `CH_3_GATE_2`.

[90] See: https://mediadl.musictribe.com/media/PLM/data/docs/P0BV2/EFFECTS%20GUIDE_M_BE_0603-AEN_WING.pdf

In the small program shown below, we replace the default `Gate` and `Dynamics` plugins for Channel 1 and set their respective parameters values to arbitrary values. For this example, we use the settings of the `AUTO RIDER DYNAMICS` gate and `Dynamics` plugins; Note that the settings are different for Gate and Compression, despite the plugin carrying the same name.

```
#include <stdio.h>
#include <string.h>
//
#include "wapi.h"
#include "wext.h"
//
int main () {

    char  wingip[24] = "";

    // we don't know the IP of our console…
    if (wOpen(wingip) != WSUCCESS) exit(1);
    printf("WING found at IP: %s\n", wingip);

    wSetTokenString(CH_1_GATE_MDL, "RIDE"); //Auto Rider Dynamics
    wSetTokenFloat(CH_1_GATE_1, -30.);      // thr
    wSetTokenFloat(CH_1_GATE_2, 0.);        // tgt
    wSetTokenInt(CH_1_GATE_3, 20);          // spd
    wSetTokenFloat(CH_1_GATE_4, 8.);        // ratio
    wSetTokenFloat(CH_1_GATE_5, 0.5);       // hold
    wSetTokenFloat(CH_1_GATE_6, 6.0);       // range

    wSetTokenString(CH_1_DYN_MDL, "RIDE");  //Auto Rider Dynamics
    wSetTokenFloat(CH_1_DYN_1, 50.);        // mix
    wSetTokenFloat(CH_1_DYN_2, 0.);         // gain
    wSetTokenFloat(CH_1_DYN_3, -30.);       // thr
    wSetTokenFloat(CH_1_DYN_4, 0.);         // tgt
    wSetTokenInt(CH_1_DYN_5, 20);           // spd
    wSetTokenFloat(CH_1_DYN_6, 4.);         // ratio
    wSetTokenFloat(CH_1_DYN_7, 0.5);        // hold
    wSetTokenFloat(CH_1_DYN_8, 3.0);        // range

    wClose();
    return 0;
}
```

Effects

Effects nodes are part of the main `JSON` structure, under the `fx.n` names, with `n: [1…16]` representing the 16 effects slots available for simultaneous use in the WIN audio processing. These 16 slots are divided in two sets of slots: 1-8 accepting premium, standard or channel effects, and slots 9-16 accepting standard and channel effects, respectively.

As in the case of plugins, the choice of effect is represented by the name (or model) of the effect, as set under the respective "`mdl`" token; authorized values are:

Premium

```
NONE, EXTERNAL, HALL REVERB, ROOM REVERB, CHAMBER REVERB, PLATE REVERB, CONCERT REVERB,
AMBIENCE, VSS3 REVERB, VINTAGE ROOM, VINTAGE REVERB, VINTAGE PLATE, BLUE PLATE, GATED
REVERB, REVERSE REVERB, DELAY/REVERB, SHIMMER REVERB, SPRING REVERB, DIMENSION CRS, STEREO
CHORUS, STEREO FLANGER, STEREO DELAY, ULTRATAP DELAY, TAPE DELAY, OILCAN DELAYB, BBD DELAY,
STEREO PITCH, DUAL PITCH
```

Standard

```
NONE, EXTERNAL, GRAPHIC EQ, PIA 560 GEQ, SPEAKER MANAGER, TRIPLE DYNAMIC EQ, C5-COMBINATOR,
PRECISION LIMITER, 2-BAND DEESSER, ULTRA ENHANCER, EXCITER, PSYCHO BASS, SUB OCTAVER, SUB
MONSTER, VELVET IMAGER, DOUBLE VOCAL, PICH FIX, ROTARY SPEAKER, PHASER, TREMOLO/PANNER,
TAPE MACHINE, MOOD FILTER, BODYREZ, RACK AMP, UK ROCK AMP, ANGEL AMP, JAZZ CLEAN AMP,
DELUXE AMP
```

Channel

```
NONE, EXTERNAL, SOUL ANALOGUE, EVEN 88 FORMANT, EVEN 84, FORTISSIMO 110, PULSAR,
MACH EQ4, EVEN CHANNEL, SOUL CHANNEL, VINTAGE CHANNEL, BUS CHANNEL, MASTERING
```

Effects can be used as dedicated inserts at defined locations within the audio path.
If an effect is part of a channel insert, assigning the effect to a different channel will remove the effect from its previous channel assignment. In order to create a more traditional effect bus, WING requires to dedicate one of the channels to the operation; Channels that want to use the effect bus can the send their audio (or a part of it) to the channel that carries the effect, creating an effect mix bus that will apply the same effect to several sources mixed into the effect channel and provide the resulting effect as a traditional effect return that can be routed to a bus.

As for the case of plugins, Effect types/engines are represented by their respective model's name under the "`mdl`" OSC tag, enabling the selection (loading) of a specific in one of the 16 available effect slots.

The `JSON` tree dedicated to effects has the following structure:

```
"fx": {
        "1": {
                "mdl": "NONE",
                "fxmix": 100
        }
        "2"…"16": {}
}
```

In fact, there are a few more, read-only[91] elements in the actual WING structure of a non-affected effect slot, resulting in the following `JSON` structure:

```
"fx": {
        "1": {
                "mdl": "NONE",
```

[91] Read-only JSON elements start with a '$' character

```
                "fxmix": 100,
                "$esrc": 0,     external source: [0…400]
                "$emode": M,    external mode: Mono, Stereo, Mid/Side
                "$a_chn": 0,    assign channel: [0…76]
                "$a_pos": 0     assign position: 0, 1]
        }
        "2"…"16": {}
}
```

Once an effect is assigned to a slot, the `JSON` structure for the respective slot is extended to include the parameters for the assigned effect. For example, installing reverb effect "`ROOM`" in effect slot 5 will result in the following update to the `JSON` of effect 5:

```
"fx": {
…
        "5": {
                "mdl": "ROOM",
                "fxmix": 100,
                "$esrc": 0,     [0…400]
                "$emode": M,    [M, ST, M/S]
                "$a_chn": 0,    [0, 1]
                "$a_pos": 0,    [0, 1]
                "pdel":         pre-delay
                "size":         room size
                "dcy":          decay
                "mult":         bass multiplier
                "damp":         damping
                "lc":           low cut
                "hc":           high cut
                "shp":          shape
                "sprd":         spread
                "diff":         diffusion
                "spin":         spin
                "ecl":          echo left
                "ecr":          echo right
                "efl":          feed left
                "efr":          feed right
        }
…
}
```

Each available effect is a sort of program including a set of dedicated parameters. When choosing a specific effect, the effect program is instantiated in one of the available slots and its parameters are mapped to the main Jason parameters lists for that effect slot, thus enabling for example up to 16 different copies[92] of the same effect to be active on every effect slot, with differentiated parameters for each slot.

The tables in "Appendix: Effects and Plugins' Parameters list, provide all effects' names and parameters, and the parameter types associated with each known effect.

Dynamic parameters anonymization in wapi

For a **wapi** program to gain access to `fx` parameters (or other dynamic parameters found in `eq`, `flt`, `dyn`, `gate`, `midi`, etc.), independently from the effect being installed/loaded at a given slot, parameter names are being 'anonymized' to names `1…40`, rather than the names that are listed with each single effect. These names listed in the tables below are preceded with their apparition number in the effect parameter list; For example, to access frequency band `125Hz` of a `Graphics EQ` effect loaded at effect slot `12`, you would set the token value to `FX_12_9`.

[92] For standard effects, 8 for premium effects

To set/instantiate an effect in one of the 16 WING FX slots, just set the effect model's name; The effect engine will be loaded to the effect slot, discarding a previous one if there was one already. The newly installed effect parameters will become available for tweaking the effect to your settings.

Model names can be found behind the tag: "`mdl`" in the tables in the "Effects and Plugins' Parameters list" appendix, further down in this document.

In the code example below is shown the instantiation of a graphic equalizer at effect slot 1, and the manipulation of its full set of parameters; In no way an interesting EQ setting, but a simple program example on how to install an effect and set parameters.

```
#include <stdio.h>
#include <string.h>
//
#include "wapi.h"
#include "wext.h"
//
int main() {
    char  wingip[24] = "";
    char *ty[] = {"std", "tru"};
    //
    // we don't know the IP of our console…
    if (wOpen(wingip)!= WSUCCESS) exit(1);
    printf("WING found at IP: %s\n", wingip);
    //
    int j = 0;
    float f = -15.;
    wSetTokenString(FX_1_MDL, "GEQ");
    while(1) {
        for (int I = FX_1_2; I <= FX_1_32; i++)
            wSetTokenFloat(I, f); // bands 30Hz to 20kHz
        Sleep(1);                     // slow down!
        F += 0.25;
        if (f > 15.) {
            f = -15;
            j ^= 1;
            wSetTokenString(FX_1_1, ty[j]);  // type
        }
    }
    wClose();
    return 0;
}
```

WING MIDI

WING MIDI (Remote-Control)

In addition to OSC and native modes, WING offers `MIDI` options and commands to remotely control the desk capabilities. Some obvious use will be for `DAW` control (see next Appendix: MCU [DAW BUTTONS] commands list), but more options for `MIDI` remote control are available:

- MIDI REMOTE CONTROL mode can be used over DIN or USB `WING MIDI control` port on the Standard WING, or `WING MIDI DAW 2` on WING Rack or Compact.
- DAW CONTROL mode can be used over `DIN` (limited to Control+Single MCU) or `USB` (Full surface) using `WING MIDI DAW 1…3` ports depending on the console type (Compact/Standard) and which DAW section MIDI data is coming from/going to.

MIDI port names

Wing Standard comes with 4 separate MIDI ports: `(WING) MIDI control, and (WING) MIDI DAW 1…3`
Wing Compact and Rack come with 2 separate MIDI ports: `(WING) MIDI DAW 1…2`
In the case of Rack and Compact models, `MIDI REMOTE CONTROL` is supported using port `(WING) MIDI DAW 2`.
The names mentioned above are for when in a Windows environment. If using MacOS, port names are `(WING) Port 1..2` on Compact and Rack

When multiple devices are connected to a same computer, MIDI port names will be differentiated using a prefix, such as `WING` or `2-WING` before them, so for example "`MIDI DAW 1`" from a Compact and a Rack will list as `WING MIDI DAW 1` and `2-WING MIDI DAW 1` when using Windows. MIDI port names are prefixed with `WING,` `WING-Compact,` or `WING-Rack` when using MacOS.

Important note on USB & MIDI: Changing the clock rate or number of USB Audio channels on WING causes USB to disconnect for a few seconds (including MIDI). On certain operating systems, this may also reset already active MIDI connections. This could happen when loading snapshots with different clock rate or USB Audio configuration.

MIDI REMOTE CONTROL[93].

MIDI commands are divided by channels, most of them have been published in a Music Group Document shown below. Additional commands via MIDI CH7, CH8 and CH9 are available for Scene recall and Show control.

CC to Channel Mapping (for FADER, MUTE, PAN). FADER on MIDI Ch 1, MUTE on MIDI Ch 2, PAN on MIDI Ch 3:

CC12..31 → Channel 1..20
CC44..6 → Channel 21..40
CC70..77 → Aux 1..8
CC78..93 → Bus 1..16
CC94/95 → Main 1..2
CC102/103 → Main 3..4
CC104..111 → Matrix 1..8
CC9: request all

[93] MIDI DAW mode is something different and is presented in a separate appendix

DCA/Mute Groups. DCA Fader on MIDI Ch 4, DCA Mute and Mute Group MUTE on MIDI Ch 5:

CC12..27 → DCA 1..16
CC28..31 → Mute Group 1..4
CC44..47 → Mute Group 5..8
CC9: request all

FX Parameter Control (FX1 on MIDI Ch 9 .. FX8 on MIDI Ch 16):

CC12 → Insert ON/OFF
CC13 → FX Mix
CC14 → FX Model
CC15..31 → FX Parameter 1..17
CC44..58 → FX Parameter 18..32

FX Parameter Control (FX9 on MIDI Ch 9 .. FX16 on MIDI Ch 16):

CC70 → Insert ON/OFF
CC71 → FX Mix
CC72 → FX Model
CC73..95 → FX Parameter 1..23
CC102..111→ FX Parameter 24..33

FX Parameters Request (MIDI Ch 9 CC9, value 0..16 for FX 1..FX16 data):

Custom Controls Remote (RX only, user layers 1..16) on MIDI Ch 6:

CC12..15 → Layer 1 Rotaries	Note0..7→ Layer 1 Buttons (upper row, lower row)
CC16..19 → Layer 2 Rotaries	Note8..15→ Layer 2 Buttons (upper row, lower row)
CC20..23 → Layer 3 Rotaries	Note16..25→ Layer 3 Buttons (upper row, lower row)
CC24.27 → Layer 4 Rotaries	Note24..31→ Layer 4 Buttons (upper row, lower row)
CC28..31 → Layer 5 Rotaries	Note32..39→ Layer 5 Buttons (upper row, lower row)
CC44..47 → Layer 6 Rotaries	Note40..47→ Layer 6 Buttons (upper row, lower row)
CC48..51 → Layer 7 Rotaries	Note48..55→ Layer 7 Buttons (upper row, lower row)
CC52..55 → Layer 8 Rotaries	Note56..63→ Layer 8 Buttons (upper row, lower row)
CC56..59 → Layer 9 Rotaries	Note64..71→ Layer 9 Buttons (upper row, lower row)
CC60..63 → Layer 10 Rotaries	Note72..79→ Layer 10 Buttons (upper row, lower row)
CC70..73 → Layer11 Rotaries	Note80..87→ Layer 11 Buttons (upper row, lower row)
CC74..77 → Layer 12 Rotaries	Note88..95→ Layer 12 Buttons (upper row, lower row)
CC78..81→ Layer13 Rotaries	Note96..103→ Layer 13 Buttons (upper row, lower row)
CC82..85 → Layer14 Rotaries	Note104..111→ Layer 14 Buttons (upper row, lower row)
CC86..89 → Layer15 Rotaries	Note112..119→ Layer 15 Buttons (upper row, lower row)
CC90..93 → Layer16 Rotaries	Note120..127→ Layer 16 Buttons (upper row, lower row)

MIDI Scene Change (on MIDI Ch 7)[94]:

CH7 CC0 (bank MSB), CH7 PC 1..128 → Scene number 1..128, number 129..256 on bank MSB 1, etc.	B60000..B60008, C600..C67F

MIDI Show Control (on MIDI Ch 8 & Ch 9):

CH8 CC0 (bank MSB), CH8 PC 1..128 → Scene Tag #1..#128 on bank MSB 0, #129..#256 on bank MSB 1, etc.	B70000..B7007F, C700..C77F
CH9 PC 1→ Scene GO	C800
CH9 PC 2→ Scene PREV	C801
CH9 PC 3→ Scene NEXT	C802
CH9 PC 4→ Scene GO PREV	C803
CH9 PC 5→ Scene GO NEXT	C804

[94] See appendix on Shows and Scenes further in this document for details on why it can be better to use scene tags rather than scene numbers when recalling show items using MIDI

What about WING faders or controls that are **not listed** in the above tables? There may be a workaround for some of them if they are controllable (i.e. listed by Behringer as such) under CC buttons or knobs; Just assign their functions to a CC button and use the MIDI code or command for that CC button! This will use a CC button or knob, but is much easier in most cases than using SYSEX, and will also/even work if your WING model doesn't have physical buttons or knobs that correspond to the used CC for your command (which is the case for rotary knobs on WING compact for example).

WING MIDI SYSEX

WING also supports MIDI SYSEX messages, part of the MIDI protocol implementation in the console. SYSEX is a key component of MIDI implementation for advanced, digital desks as many commands are dedicated to controlling the desk as a surface control, rather than sending MIDI instrument notes. Standard 3 bytes MIDI messages are generally not long enough to support the full set of capabilities these new desks offer. This is made possible through the use and support of SYSEX functionality.

MIDI SYSEX messages are "system exclusive" data that can be passed using the MIDI HW and standard protocol to the console, using a specific formatting convention and system dedicated messages, sent over MIDI.

Your WING should be set to accept SYSEX data over `USB` or `DIN`. This setting is part of the in the `MIDI REMOTE CONTROL` tab in the `SETUP→REMOTE` screen.

SYSEX Messages format

The formatting used in SYSEX data is similar to the one used for Node Data: each parameter group is separated by a ',' character, the '/' character represents the root of the `JSON` parameter tree, and '.' characters are used to navigate up and down within the `JSON` parameter tree, as shown below:

```
/ch.1.fdr=-1,mute=0,.2.fdr=0,mute=1
```

will set channel 1 fader to -1dB, will unmute the channel, and set channel 2 fader to 0dB and mute the channel.

SYSEX communications can report error messages (see below) that can be:

```
NODE NOT FOUND
VALUE ERROR
BUFFER OVERFLOW
NODE IS NOT PAR
INCOMPLETE DATA
STACK EMPTY
```

SYSEX Messages, Explained

WING MIDI SYSEX messages are built as follows:

```
F0 00 20 32 57 <cmd> <data[*]> F7
```

`<data[*]>` is an arbitrary number of bytes representing the data to be sent to the console, and can be an empty string of bytes; The way data is interpreted by the console is controlled using `<cmd>`, a single byte as listed below:

`00`: ident

Will return the signature string of the console, such as returned with OSC command `/?`[95] sent to port 2223, or native identification datagram `WING?`[96] Sent to port 2222. Upon executing the command, the console will return a MIDI byte `01`.

`02`: execute `<data[*]>`

Will execute the requested command contained in or represented by `<data[*]>`. `<data[*]>` should contain a WING command respecting the format rules described earlier in this chapter. Upon executing the command, the console may return status `07 (Error)` followed with an error message if an error occurred.

`03`: dump `<data[*]>`

Will return the current values found for `<data[*]>`. `<data[*]>` should contain a WING node or command respecting the format rules described earlier in this chapter. Upon executing the command, the console will return MIDI byte `04` (OK) or `07 (Error)` followed with an error message.

`05`: describe `<data[*]>`

Will return the description of the node found at or represented by `<data[*]>`. `<data[*]>` should contain a WING node respecting the format rules described earlier in this chapter. Upon executing the command, the console will return a MIDI byte `06` (OK) or `07` followed with an error message.

Examples

We give below a few examples of MIDI SYSEX commands sent to WING, with their interpretation, their resulting effect on WING, and the returned data, if any.

cmd = 00 example:

Sending `F0 00 20 32 57 00 f7`,
or `cmd 00` will generate a SYSEX being returned by WING:

```
F0 00 20 32 57 01 57 49 4E 47 2C 31 39 32 2E 31 36 38
2E 31 2E 37 31 2C 50 47 4D 2C 6E 67 63 2D 66 75 6C 6C
2C 4E 4F 5F 53 45 52 49 41 4C 2C 31 2E 30 38 2E 31 2D
30 2D 67 33 33 65 36 39 66 38 38 3A 72 65 6C 65 61 73
65 F7
```

Containing the WING signature *(your system will contain something slightly different)*:
`WING,192.168.1.71,PGM,ngc-full,NO_SERIAL,1.08.1-0-g33e69f88:release`

cmd = 02 examples:

Sending `F0 00 20 32 57 02 2F 63 68 2E 31 2E 66 64 72 3D 2D 31 2C 6D 75 74 65 3D 30 2C 2E 32 2E 66 64 72 3D 30 2C 6D 75 74 65 3D 31 F7`,
or `cmd 02` followed by `/ch.1.fdr=-1,mute=0,.2.fdr=0,mute=1` will set channel 1 fader to -1, channel 2 fader to 0 and will unmute channel 1 and mute channel 2.

[95] Refer to chapter on OSC protocol
[96] Refer to chapter "Remote communications with WING"

Sending F0 00 20 32 57 02 2F 63 68 2E 31 2E 66 64 3D 2D 31 2C 6D 75 74 65 3D 30 2C 2E 32 2E 66 64 72 3D 30 2C 6D 75 74 65 3D 31 F7,
or cmd 02 followed by /ch.1.fd=-1,mute=0,.2.fdr=0,mute=1 will return an error; note we omitted the "r" of "fdr" above for channel 1. WING will reply with the following SYSEX message: F0 00 20 32 57 07 4E 4F 44 45 20 4E 4F 54 20 46 4F 55 4E 44 F7, or error status 07, followed by NODE NOT FOUND.

cmd = 03 examples:

Sending F0 00 20 32 57 03 2f 61 75 78 F7,
or cmd 03 followed by /aux, will be replied by WING with a near 28000 bytes SYSEX message corresponding to the following in ASCII (partial listing containing aux1 and aux8, aux2 to aux7 included but not listed):

```
1.in.set.srcauto=0,altsrc=0,inv=0,trim=0.0,bal=0.0,.conn.grp=USB,in=1,altgrp=OFF,altin=1,..
clink=0,col=8,name=USB,icon=605,led=1,mute=0,fdr=oo,pan=0,wid=100,solosafe=0,mon=A,eq.on=0,
mix=100,lg=0.0,lf=80.2,lq=2.00,leq=SHV,1g=0.0,1f=399.1,1q=2.00,2g=0.0,2f=2k50,2q=2.00,hg=0.
0,hf=11k99,hq=2.00,heq=SHV,.preins.on=0,ins=NONE,.main.1.on=1,lvl=0.0,.2.on=0,lvl=0.0,.3.on
=0,lvl=0.0,.4.on=0,lvl=0.0,..send.1.on=0,lvl=oo,pon=0,ind=0,mode=PRE,plink=0,pan=0,wid=100,
.2.on=0,lvl=oo,pon=0,ind=0,mode=PRE,plink=0,pan=0,wid=100,.3.on=0,lvl=oo,pon=0,ind=0,mode=P
RE,plink=0,pan0,wid=100,.4.on=0,lvl=oo,pon=0,ind=0,mode=PRE,plink=0,pan=0,wid=100,.5.on=0,l
vl=oo,pon=0,ind=0,mode=PRE,plink=0,pan=0,wid=100,.6.on=0,lvl=oo,pon=0,ind=0,mode=PRE,plink=
0,pan=0,wid=100,.7.on=0,lvl=oo,pon=0,ind=0,mode=PRE,plink=0,pan=0,wid=00,.8.on=0,lvl=oo,pon
=0,ind=0,mode=PRE,plink=0,pan=0,wid=100,.9.on=0,lvl=oo,pon=0,ind=0,mode=PRE,plink=0,pan=0,w
id=100,.10.on=0,lvl=oo,pon=0,ind=0,mode=PRE,plink=0,pan=0,wid=100,.11.on=0,lvl=oo,pon=0,ind
=0,mode=PRE,plink=0,pan=0,wid=100,.12.on=0,lvl=oo,pon=0,ind=0,mode=PRE,plink=0,pan=0,wd=100
,.13.on=0,lvl=oo,pon=0,ind=0,mode=PRE,plink=0,pan=0,wid=100,.14.on=0,lvl=oo,pon=0,ind=0,mod
e=PRE,plink=0,pan=0,wid=100,.15.on=0,lvl=oo,pon=0,ind=0,mode=PRE,plink=0,pan=0,wid=100,.16.
on=0,lvl=oo,pon=0,ind=0,mode=PRE,plink=0,pan=0,wid100,..tags=,.2.in.set.srcauto=0,[…Aux cha
nnels 2 to 7 listed…],..tags=,.8.in.set.srcauto=0,altsrc=0,inv=0,trim=0.0,bal=0.0,.conn.grp
=OFF,in=1,altgrp=OFF,altin=1,..clink=0,col=1,name=,icon=0,led=1,mute=0,fdr=oo,pan=0,wid=100
,solosafe=0,mon=A,eq.on=0,mix=100,lg=0.0,lf=80.2,lq=2.00,leq=SHV,1g=0.0,1f=399.1,1q=2.00,2g
=0.0,2f=2k50,2q=2.00,hg=0.0,hf=11k99,hq=2.00,heq=SHV,.preins.on=0,ins=NONE,.main.1.on=1,lvl
=0.0,.2.on=0,lvl=0.0,.3.on=0,lvl=0.0,.4.on=0,lvl=0.0,..send.1.on=0,lvl=oo,pon=0,ind=0,mode=
PRE,plink=0,pan=0,wid=100,.2.on=0,lvl=oo,pon=0,ind=0,mode=PRE,plink=0,pan=0,wid=100,.3.on=0
,lvl=oo,pon=0,ind=0,mode=PRE,plink=0,pan=0,wid=100,.4.on=0,lvl=oo,pon=0,ind=0,mode=PRE,plin
k=0,pan=0,wid=100,.5.on=0,lvl=oo,pon=0,ind=0,mode=PRE,plink=0,pan=0,wid=100,.6.on=0,lvl=oo,
pon=0,ind=0,mode=PRE,plink=0,pan=0,wid=100,.7.on=0,lvl=oo,pon=0,ind0,mode=PRE,plink=0,pan=0
,wid=100,.8.on=0,lvl=oo,pon=0,ind=0,mode=PRE,plink=0,pan=0,wid=100,.9.on=0,lvl=oo,pon=0,ind
=0,mode=PRE,plink=0,pan=0,wid=100,.10.on=0,lvl=oo,pon=0,ind=0,mode=PRE,plink=0,pan=0,wid=10
0,.11.on=0,lvl=oo,pon=0,ind=0,mode=PRE,plink=0,pan=0,wid=100,.12.on=0,lvl=oo,pon=0,ind=0,mo
de=PRE,plink=0,pan=0,wid=100,.13.on=0,lvl=oo,pon=0,ind=0,mode=PRE,plink=0,pan=0,wid=100,.14
.on=0,lvl=oo,pon=0,ind=0,mode=PRE,plink=0,pan=0,wid=100,.15.on=0,lvl=oo,pon=0,ind=0,mode=PR
E,plink=0,pan=0,wid=100,.16.on=0,lvl=oo,pon=0,ind=0,mode=PRE,plink=0,pan=0,wid=100,..tags=,
.
```

Sending F0 00 20 32 57 03 2F 63 68 2E 31 2E 66 64 72 F7,
or cmd 03 followed by /ch.1.fdr, will be replied by WING with a SYSEX message containing the current value of channel 1 fader: F0 00 20 32 57 04 2D 31 2E 30 F7, or status 04 (no error) and value -1.0.

cmd = 05 examples:

Sending again SYSEX F0 00 20 32 57 05 2f 61 75 78 00 00 00 00 F7,
Or cmd 05 followed by /aux~~~~ will be replied by WING with the following SYSEX message (note the status 06 (no error):

```
F0 00 20 32 57 06 20 20 31 20 20 20 20 20 20 20 20 20
```

```
20 20 20 20 20 28 6E 6F 64 65 29 0A 20 20 32 20 20 20
20 20 20 20 20 20 20 20 20 20 20 28 6E 6F 64 65 29 0A
20 20 33 20 20 20 20 20 20 20 20 20 20 20 20 20 20 28
6E 6F 64 65 29 0A 20 20 34 20 20 20 20 20 20 20 20 20
20 20 20 20 20 28 6E 6F 64 65 29 0A 20 20 35 20 20 20
20 20 20 20 20 20 20 20 20 20 20 28 6E 6F 64 65 29 0A
20 20 36 20 20 20 20 20 20 20 20 20 20 20 20 20 20 28
6E 6F 64 65 29 0A 20 20 37 20 20 20 20 20 20 20 20 20
20 20 20 20 20 28 6E 6F 64 65 29 0A 20 20 38 20 20 20
20 20 20 20 20 20 20 20 20 20 20 28 6E 6F 64 65 29 0A
F7
```

Or in ASCII:

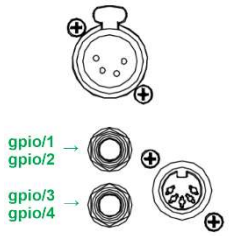
```
1               (node)
2               (node)
3               (node)
4               (node)
5               (node)
6               (node)
7               (node)
8               (node)
```

Sending again `F0 00 20 32 57 05 2F 63 68 2E 31 2E 66 64 72 F7`,
or `cmd 05` followed by `/ch.1.fdr`, will be replied by WING with a SYSEX message containing the description of channel 1 fader: `F0 00 20 32 57 06 66 61 64 65 72 20 5B 2D 6F 6F 20 2E 2E 20 31 30 2E 30 20 64 42 5D 2C 20 31 30 32 34 20 73 74 65 70 73 0A F7`, or status `06` (no error), followed by description `fader [-oo .. 10.0 dB], 1024 steps`.

Appendices

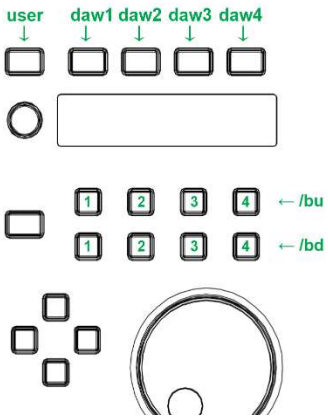
Appendix: Buttons (user/gpio, user/user, user/daw, user/)[97]

WING includes a rather large set of buttons separated in different logical blocks: user/gpio, user/user, and user/daw and user. They are all managed under the `$ctl` subtree of commands. As in the case of effects where the effect model sets the type and number of OSC patterns available for supporting the functionality currently in effect, the associated `JSON` structure varies and adapts to the necessary sets of parameters.



user/gpio/1..4

This subsection covers the 4 possible GPIOs supported by WING; the actual set of usable OSC patterns available at a given time depends on the `mode` parameter value of the `/$ctl/user/gpio/1..4/bu/` OSC pattern represented below as `<OSC b_pattern>`.



user/user/1..4

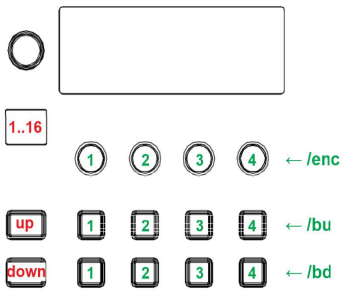
This subsection covers the 8 user buttons supported by WING[98]; the actual set of usable OSC patterns available at a given time depends on the `mode` parameter value of the `/$ctl/user/user/1..4/bu/` and the `/$ctl/user/user/1..4/bd/` OSC patterns for the 4 buttons of the upper and lower row of the button section, and represented below as `<OSC b_pattern>`.

user/daw1..4/1..4

This subsection covers the 4 possible sets of 8 DAW buttons supported by WING; the actual set of usable OSC patterns available at a given time depends on the `mode` parameter value of the `/$ctl/user/daw1..daw4/1..4/bu/` and the `/$ctl/user/daw1..daw4/1..4/bd/` OSC patterns for the 4 buttons of the upper and lower row of the button section, and represented below as `<OSC b_pattern>`.

[97] Describing here the buttons and CC section for the standard WING, Compact and Rack version differ in the amount of CC buttons/knobs available.
[98] Standard model only

user/1..16/1..4



This subsection covers the 16 possible sets of 8 user buttons and 4 user encoders supported by WING[99]; the actual set of usable OSC patterns available at a given time depends on the `mode` parameter value of the `/$ctl/user/1..16/1..4/bu/`, `/$ctl/user/1..16/1..4/bd/`, and `/$ctl/user/1..16/1..4/enc/` OSC patterns for the 4 buttons of the upper and lower row of the button section, and the 4 encoders represented below as `<OSC b_pattern>` and `<OSC e_pattern>`, respectively.

[99] Standard and Rack models only

The tables below list the different options for OSC patterns `<OSC b_pattern>`:

mode	Command	Type	Range / Text	Description
OFF	none			OFF
MUTE	<OSC b_pattern>/ch	I	1..76	Channel number
INS1	<OSC b_pattern>/ch	I	1..76	Channel number
INS2	<OSC b_pattern>/ch	I	1..76	Channel number
MGRP	<OSC b_pattern>/mgrp	S	MGRP.1, MGRP.2,..,MGRP.8	Mute group number
DCAMUTE	<OSC b_pattern>/dca	S	DCA.1, DCA.2,..,DCA.16	DCA fader number mute
SOF	<OSC b_pattern>/ch	I	1..76	Channel number
SPILL	<OSC b_pattern>/area	S	L, C, R	Console area (left, center, right)
	<OSC b_pattern>/tgt	S	DCA1,.., DCA16, FX1,.., FX16, BUS1,.., BU16, MAIN1,.., MAIN4, MTX1,..,MTX8, AUTOX, AUTOY	Targeted group[100]
FXPAR	<OSC b_pattern>/fx	S	FX1,..,FX16	FX processor number
	<OSC b_pattern>/par	I	1..41	FX processor parameter number[101]
DAWBTN	<OSC b_pattern>/btn	S	T1,..,T20, N1,..,N9, A1,..,A16, F1,..,F8, V1,..,V15, AU1,..,AU12, SY1,..,SY10, OT1,..,OT12, E1,..,E10, SP1,.., SP6	MCU button[102]
DAWENC	<OSC b_pattern>/enc	S	M1P,..,M8P, E1P,..,E16P, M1,..,M8, E1,..,E16, JOG	DAW Rotary[103]
CHPAGE	<OSC b_pattern>/ch	I	1..76	Channel number
	<OSC b_pattern>/pg	S	HOME, INPUT, FILT, GATE, DYN, EQ, INS1, INS2, MAIN, SEND, SND.EQ	Page name
PAGE	<OSC b_pattern>/pg	S	FX, MTRS, CHINS, SRC, OUTS, SETUP, LIB, CUSTCTL, MON, 2TRK, WLIVE, MIXV, FDRV, SENDV, MGRP, LAYER	Page name
FDRPAGE	<OSC b_pattern>/area	S	L, C, R, CMPCT, RCK, EXT, VRT	Area
	<OSC b_pattern>/bank	I	1..21	Bank

[100] Maps to /$ctl/layer/xxx/spidx parameters values 1..62
[101] 1..40 are for FX parameters, 41 is for FX Mix
[102] See MCU [DAW BUTTONS] commands list in Appendixes
[103] See MCU [DAW V-POTS] commands list in Appendixes

VIEWPAGE	<OSC b_pattern>/area	S	L, C, R, CMPCT, RCK, EXT, VRT	Area
	<OSC b_pattern>/bank	I	1..21	Bank
OTHER	<OSC b_pattern>/other	S	TBA, TBB, ALTSRC, DAWSW, MONA, MONB, MONAB, MONDIM, MONMONO, MONSWAP, MONMUTE, FDROFF, FDR-10DB, FDR0DB, AUTOX, AUTOY, CHPREV, CHNEXT, CHSOLO, BUSSOLO, MAINSOLO, MTXSOLO, CHMTR, BUSMTR, MAINMTR, MTXMTR, DCAMTR, CCBANK	Other functions
	<OSC b_pattern>/CCBK	S	1..16, -1, +1	When other == CCBANK
GPIO	<OSC b_pattern>/GPIO	S	A, B, C, D, A-P, B-P, C-P, D-P, 2S, 5S, 10S, 15S, 20S, 25S, 30S	GPIO Toggle, Push, or Power-on delay[104]
FSTART	<OSC b_pattern>/ch	I	1..76	Channel number
MIDICCT	<OSC b_pattern>/ch	I	1..16	MIDI channel (toggle)
	<OSC b_pattern>/cc	I	0..127	MIDI control change number
	<OSC b_pattern>/val	I	0..127	MIDI control value
MIDICCP	<OSC b_pattern>/ch	I	1..16	MIDI channel (push)
	<OSC b_pattern>/cc	I	0..127	MIDI control change number
	<OSC b_bpattern>/val	I	0..127	MIDI control value
MIDINT	<OSC b_pattern>/ch	I	1..16	MIDI channel (toggle)
	<OSC b_bpattern>/note	I	0..127	MIDI note
	<OSC bb_pattern>/val	I	0..127	MIDI note value
MIDINP	<OSC b_pattern>/ch	I	1..16	MIDI channel (push)
	<OSC b_pattern>/note	I	0..127	MIDI note
	<OSC b_pattern>/val	I	0..127	MIDI note value
MIDIPGM	<OSC b_pattern>/ch	I	1..16	MIDI channel
	<OSC b_pattern>/note	I	1..128	MIDI program value
USBPR	<OSC b_pattern>/usbpr	S	PSTOP, PLAY, PPAUSE, PNEXT, PPREV, RSTOP, RECORD, RPAUSE, RNEW	USB Play Rec
SDRECA	<OSC b_pattern>/sdrec	S	STOP, PLAY, REC, PAUSE, PLAYSTOP, PLAYPAUSE, ADD, PREV, NEXT, PLAYMARKER, GOMARKER, SELSESSION, PREV_S, NEXT_S	SD A recorder

[104] Toggle (A..D) and Push (A-P..D-P) apply to GPIO and USER CC; Delays {2S…30S) apply only to GPIO state setup time after a console power cycle.

SESSIONA	<OSC b_pattern>/session	S	S1..S20	SD A Session
MARKERA	<OSC b_pattern>/marker	S	M1..M20	SD A Marker
SDRECB	<OSC b_pattern>/sdrec	S	STOP, PLAY, REC, PAUSE, PLAYSTOP, PLAYPAUSE, ADD, PREV, NEXT, PLAYMARKER, GOMARKER, SELSESSION, PREV_S, NEXT_S	SD B recorder
SESSIONB	<OSC b_pattern>/session	S	S1..S20	SD B Session
MARKERB	<OSC b_pattern>/marker	S	M1..M20	SD B Marker

The table below lists the different options for the OSC encoder pattern `<OSC e_pattern>`:

mode	Command	Type	Range / Text	Description
OFF	none			OFF
FDR	<OSC e_pattern>/ch	I	1..76	Channel number
PAN	<OSC e_pattern>/ch	I	1..76	Channel number
DCA	<OSC e_pattern>/dca	S	DCA.1, DCA.2,..,DCA.16	DCA fader number
SSND	<OSC e_pattern>/send	S	BUS1,..,BUS16, MAIN1,..,MAIN4, MTX1,..,MTX8	Send to Bus, Main or Matrix number
FSND	<OSC e_pattern>/ch <OSC e_pattern>/send	I S	1..48 BUS1,..,BUS16, MAIN1,..,MAIN4, MTX1,..,MTX8	Channel number Send to Bus, Main or Matrix number
FX	<OSC e_pattern>/fx	S	FX1,..,FX16	FX processor number
	<OSC e_pattern>/par	I	1..41	FX processor parameter number[105]
DAWMCU	<OSC e_pattern>/mcuenc	S	M1,..,M8, E1,..,E16, JOG	DAW Rotary[106]
MON	<OSC e_pattern>/mon	S	A, B	Monitor selection[107] A: PHONES level B: SPEAKERS / MONITOR level

[105] 1..40 are for FX parameters, 41 is for FX Mix

[106] See MCU [DAW REMOTE MCU] commands list in Appendixes

[107] Note these are only 'readable' when touching the encoder, the actual setting is only possible with the potentiometer in the Monitoring/Talkback section of the console

OTHER	<OSC e_pattern>/other	S	BRILAMP, BRIGLOW, BRIPATCH	Lamp and other lighting controls
MIDICC	<OSC e_pattern>/ch	I	1..16	MIDI channel
	<OSC e_pattern>/cc	I	0..127	MIDI control change number
	<OSC e_pattern>/val	I	0..127	MIDI control change value
SD A	<OSC e_pattern>/sdarec	S	POS, MARKER, SESSION	SD-A Recorder
SD B	<OSC e_pattern>/sdbrec	S	POS, MARKER, SESSION	SD-B Recorder

Appendix: Effects and Plugins' Parameters list

In the (long) tables below, we list all known/exposed effects and plugins available with the WING digital console, along with their name, type, and min/max/step/list values; We therefore present Standard Effects, Premium effects, Filter Plugins, Gate Plugins, EQ Plugins, and Compressor Plugins.
All active effects and plugins modify the `JSON` tree and their respective OSC patterns.

On any channel, an insert will create a processing delay of 32 samples (i.e. around 0.66ms). This is because audio is routed to a different DSP for FX processing. It is important to take this into account when mixing, as phasing effects may result from the imposed delay.

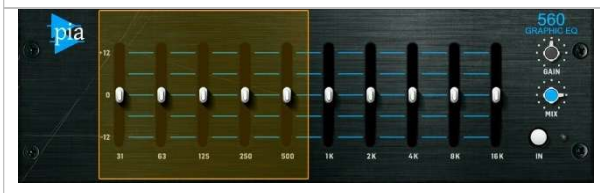
With FW 2.1, Behringer published a document on effect: the WING Effects Guide, with a description of effects and plugins that can be found on WING. This document can be found and downloaded at: https://mediadl.musictribe.com/media/PLM/data/docs/P0BV2/EFFECTS%20GUIDE_M_BE_0603-AEN_WING.pdf

In addition to the Behringer document above, the tables below show all parameters associated to effects and plugins, including their name, type, and value range following the OSC pattern `/fx/1..16/`

Effects

Standard effects

	None *0 **"mdl"**: NONE*
	*External* *0 **"mdl"**: EXT* *1 **"egrp"**: str [OFF, LCL, AUX, A, B, C, SC, USB, CRD, MOD, PLAY, AES] ext grp* *2 **"ein"**: int [1…64] ext in* *3 **"emode"**: str [M, ST, M/S] ext mode* *4 **"lat"**: int [0…200] latency* *5 **"trim"**: linf [-18, 18, 361] dB, trim*
	*Graphic EQ* *0 **"mdl"**: GEQ* *1 **"type"**: str [STD, TRU] geq type* *2 **"20"**: linf [-15, 15, 121] dB* *3 **"25"**: linf [-15, 15, 121] dB* *4 **"31"**: linf [-15, 15, 121] dB* *5 **"40"**: linf [-15, 15, 121] dB* *6 **"50"**: linf [-15, 15, 121] dB* *7 **"63"**: linf [-15, 15, 121] dB* *8 **"80"**: linf [-15, 15, 121] dB* *9 **"100"**: linf [-15, 15, 121] dB* *10 **"125"**: linf [-15, 15, 121] dB* *11 **"160"**: linf [-15, 15, 121] dB* *12 **"200"**: linf [-15, 15, 121] dB* *13 **"250"**: linf [-15, 15, 121] dB* *14 **"315"**: linf [-15, 15, 121] dB* *15 **"400"**: linf [-15, 15, 121] dB* *16 **"500"**: linf [-15, 15, 121] dB* *17 **"630"**: linf [-15, 15, 121] dB* *18 **"800"**: linf [-15, 15, 121] dB* *19 **"1k"**: linf [-15, 15, 121] dB*

	20 **“1k25”**: linf [-15, 15, 121] dB 21 **“1k6”**: linf [-15, 15, 121] dB 22 **“2k”**: linf [-15, 15, 121] dB 23 **“2k5”**: linf [-15, 15, 121] dB 24 **“3k15”**: linf [-15, 15, 121] dB 25 **“4k”**: linf [-15, 15, 121] dB 26 **“5k”**: linf [-15, 15, 121] dB 27 **“6k3”**: linf [-15, 15, 121] dB 28 **“8k”**: linf [-15, 15, 121] dB 29 **“10k”**: linf [-15, 15, 121] dB 30 **“12k5”**: linf [-15, 15, 121] dB 31 **“16k”**: linf [-15, 15, 121] dB 32 **“20k”**: linf [-15, 15, 121] dB 33 **“TRIM”**: linf [-15, 15, 121] dB
	PIA 560 GEQ 0 **“mdl”**: PIA 1 **“mix”**: linf [0, 125, 126] %, mix 2 **“gain”**: linf [-12, 12, 241] dB 3 **“31”**: linf [-12, 12, 241] dB 4 **“63”**: linf [-12, 12, 241] dB 5 **“125”**: linf [-12, 12, 241] dB 6 **“250”**: linf [-12, 12, 241] dB 7 **“500”**: linf [-12, 12, 241] dB 8 **“1k”**: linf [-12, 12, 241] dB 9 **“2k”**: linf [-12, 12, 241] dB 10 **“4k”**: linf [-12, 12, 241] dB 11 **“8k”**: linf [-12, 12, 241] dB 12 **“16k”**: linf [-12, 12, 241] dB
	Triple Dynamic EQ 0 **“mdl”**: DEQ3 1 **“1-thr”**: linf [-60, 0, 121] dB, threshold 1 2 **“1-ratio”**: str [1.20, 1.30, 1.50, 2.00, 3.00, 5.00, 10.00] ms, ratio 1 3 **“1-att”**: linf [0.00, 200.00, 201] ms, att 1 4 **“1-rel”**: logf [20.00, 4000.00, 130] ms, rel 1 5 **“1-filt”**: str [OFF, BP, LP6, LP12, HP6, HP12] 6 **“1-g”**: linf [-15.00, 15.00, 301] dB, gain 1 7 **“1-f”**: logf [20, 20000, 961] Hz, freq 1 8 **“1-q”**: logf [0.44, 10.00, 181] qual 1 9 **“1-mode”**: str [low, high] mode 1 10 **“2-thr”**: linf [-60, 0, 121] dB, threshold 2 11 **“2-ratio”**: str [1.20, 1.30, 1.50, 2.00, 3.00, 5.00, 10.00] ms, ratio 2 12 **“2-att”**: linf [0.00, 200.00, 201] ms, att 2 13 **“2-rel”**: logf [20.00, 4000.00, 130] ms, rel 2 14 **“2-filt”**: str [OFF, BP, LP6, LP12, HP6, HP12] 15 **“2-g”**: linf [-15.00, 15.00, 301] dB, gain 2 16 **“2-f”**: logf [20, 20000, 961] Hz, freq 2 17 **“2-q”**: logf [0.44, 10.00, 181] qual 2 18 **“2-mode”**: str [low, high] mode 2 19 **“3-thr”**: linf [-60, 0, 121] dB, threshold 3 20 **“3-ratio”**: str [1.20, 1.30, 1.50, 2.00, 3.00, 5.00, 10.00] ms, ratio 3 21 **“3-att”**: linf [0.00, 200.00, 201] ms, att 3 22 **“3-rel”**: logf [20.00, 4000.00, 130] ms, rel 3 23 **“3-filt”**: str [OFF, BP, LP6, LP12, HP6, HP12] 24 **“3-g”**: linf [-15.00, 15.00, 301] dB, gain 3 25 **“3-f”**: logf [20, 20000, 961] Hz, freq 3 26 **“3-q”**: logf [0.44, 10.00, 181] qual 3 27 **“3-mode”**: str [low, high] mode 3

Combinator

```
 0 "mdl":       C5-CMB
 1 "thr":       linf [-40, 0, 401] dB, threshold
 2 "gain":      linf [-10, 10, 201] dB, gain
 3 "ratio":     str [1.1, 1.2, 1.3, 1.5, 1.7,
                     2.0, 2.5, 3.0, 3.5, 4.0,
                     5.0, 7.0, 10.0, 100.0] ms, ratio
 4 "slope":     str [24, 48] dB/Oct, slope
 5 "bandse l":  int [1..5] selected band
 6 "att":       linf [0, 20, 21] attack
 7 "rel":       logf [20, 3000, 201] ms, release
 8 "arel":      int [0, 1] auto release
 9 "sbc":       linf [1, 10, 10] sbc speed
10 "sbcon":     int [0, 1] sbc on
11 "thr_1":     linf [-10, 10, 201] dB, 1-THR
12 "thr_2":     linf [-10, 10, 201] dB, 2-THR
13 "thr_3":     linf [-10, 10, 201] dB, 3-THR
14 "thr_4":     linf [-10, 10, 201] dB, 4-THR
15 "thr_5":     linf [-10, 10, 201] dB, 5-THR
16 "gain_1":    linf [-10, 10, 201] dB, 1-GAIN
17 "gain_2":    linf [-10, 10, 201] dB, 2-GAIN
18 "gain_3":    linf [-10, 10, 201] dB, 3-GAIN
19 "gain_4":    linf [-10, 10, 201] dB, 4-GAIN
20 "gain_5":    linf [-10, 10, 201] dB, 5-GAIN
21 "byp_1":     int [0, 1], 1-BYP
22 "byp_2":     int [0, 1], 2-BYP
23 "byp_3":     int [0, 1], 3-BYP
24 "byp_4":     int [0, 1], 4-BYP
25 "byp_5":     int [0, 1], 5-BYP
26 "width_1":   linf [-50, 50, 101], 1-XOVER
27 "width_2":   linf [-50, 50, 101], 2-XOVER
28 "width_3":   linf [-50, 50, 101], 3-XOVER
29 "width_4":   linf [-50, 50, 101], 4-XOVER
30 "width_5":   linf [-50, 50, 101], 5-XOVER
31 "mix":       linf [0, 100, 101], mix
32 "$bdsolo":   int [0, 1] band solo
```

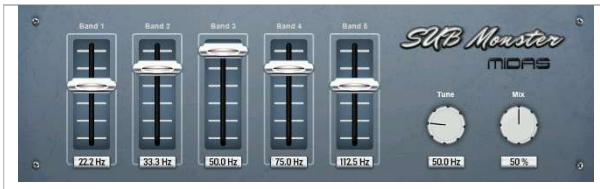
Precision Limiter

```
 0 "mdl":       LIMITER
 1 "gin":       linf [0, 18, 73] dB, in gain
 2 "gout":      linf [-18, 0, 73] dB out gain
 3 "sqz":       int [0…100] squeeze
 4 "knee":      int [0…10] knee
 5 "again":     int [0, 1] auto gain
 6 "att":       linf [.05, 1, 95] ms, attack
 7 "rel":       logf [20, 2000, 101] ms, release
```

Speaker Manager

```
 0 "mdl":       SPKMAN
 1 "hpf":       logf [20.00, 20000.00, 961] Hz, high
 2 "hptype":    str [FLAT, BW6, BW12, BS12, LR12,
                     BW18, BW24, BS24, LR24, BW48,
                     LR48] type
 3 "lpf":       logf [20.00, 20000.00, 961] Hz, low
 4 "lptype":    str [FLAT, BW6, BW12, BS12, LR12,
                     BW18, BW24, BS24, LR24, BW48,
                     LR48] type
 5 "tiltf":     logf [100.00, 10000.00, 121] Hz, tilt
 6 "tiltg":     linf [-6.00, 6.00, 121] dB, tilt gain
 7 "phase":     linf [0.00, 180.00, 37] phase
 8 "invert ":   int [0, 1] invert
 9 "dist":      linf [0.00, 5.00, 501] mtrs, distance
10 "pos":       linf [-5.00, 5.00, 1001] mtrs, pos.
11 "dyneq":     int [0, 1] deq
12 "dynthr":    linf [-60.00, 0.00, 121] dB, threshold
13 "deqratio":  str [1.20, 1.30, 1.50, 2.00, 3.00,
                     5.00, 10.00] ratio
14 "deqatt":    linf [0.00, 200.00, 201] ms, attack
```

	15 **“deqrel”**: logf [20.00, 4000.00, 130] ms, release 16 **“deqfilt”**: str [OFF, BP, LP6, LP12, HP6, HP12] 17 **“deqg”**: linf [-15.00, 15.00, 301] dB, gain 18 **“deqf”**: logf [20.00, 20000.00, 961] Hz, freq 19 **“deqq”**: logf [0.44, 10.00, 181] qual 20 **“deqmode”**: str [low, high] mode 21 **“lim”**: int [0, 1] limiter 22 **“limthr”**: linf [-24.00, 0.00, 241] dB, threshold 23 **“limrms”**: int [0, 1] rms 24 **“limrel”**: logf [50.00, 2000.00, 121] ms, release
	2-Band DeEsser 0 **“mdl”**: DE-S2 1 **“lo”**: linf [0, 50, 51] low 2 **“hi”**: linf [0, 50, 51] high 3 **“los”**: linf [0, 50, 51] low (s) 4 **“his”**: linf [0, 50, 51] high (s) 5 **“gdr”**: str [FEMALE, MALE] gender 6 **“mode”**: str [STEREO, MID/SIDE] mode
	Ultra Enhancer 0 **“mdl”**: ENHANCE 1 **“stlv”**: linf [-100, 100, 201] %, st lvl 2 **“lmf”**: linf [-100, 100, 201] %, lmf spread 3 **“lmvl”**: linf [-100, 100, 201] %, mono lvl 4 **“st”**: linf [-100, 100, 201] %, st pan 5 **“m”**: linf [-100, 100, 201] %, mono pan 6 **“bass”**: linf [0, 100, 101] %, bass gain 7 **“mid”**: linf [0, 100, 101] %, mid gain 8 **“high”**: linf [0, 100, 101] %, high gain 9 **“g”**: linf [-112, 12, 241] dB, gain 10 **“solo”**: int [0, 1] solo 11 **“bassf”**: linf [1, 50, 50] bass freq 12 **“midq”**: linf [1, 50, 50] mid Q 13 **“highf”**: linf [1, 50, 50] high freq
	Exciter 0 **“mdl”**: EXCITER 1 **“tune”**: logf [1000, 10000, 51] Hz, tune 2 **“peak”**: linf [0, 100, 101] %, peak 3 **“zfill”**: linf [0, 100, 101] %, zfill 4 **“timbre”**: linf [-50, 50, 101] timbre 5 **“harm”**: linf [0, 100, 101] %, harm 6 **“mix”**: linf [0, 100, 101] %, mix 7 **“dry”**: int [0, 1] dry
	Psycho Bass 0 **“mdl”**: P-BASS 1 **“int”**: linf [-24, 6, 61] dB, intensity 2 **“bass”**: linf [-60, 0, 121] dB, bass gain 3 **“xf”**: logf [32, 200. 51] Hz, X/O freq 4 **“solo”**: int [0, 1] solo
	Sub Octaver 0 **“mdl”**: SUB 1 **“rng”**: str [LOW, MID, HIGH] range 2 **“oct1”**: linf [0, 100, 101] %, octave 1 3 **“oct2”**: linf [0, 100, 101] %, octave 2



Sub Monster

```
0 "mdl":   SUB-M
1 "mix":   [0, 100, 101] %mix
2 "freq":  linf [45, 67.5, 226] Hz, freq
3 "bd1":   linf [0, 100, 101] %, band 1
4 "bd2":   linf [0, 100, 101] %, band 2
5 "bd3":   linf [0, 100, 101] %, band 3
6 "bd4":   linf [0, 100, 101] %, band 4
7 "bd5":   linf [0, 100, 101] %, band 5
```

Velvet Imager

```
0 "mdl":   V_IMG
1 "wid":   linf> [-1.00, 1.00, 201] width
2 "st":    linf [0.00, 100.00, 101] %, stereo
3 "gain":  linf [-6.00, 6.00, 49] dB, gain
4 "mode":  str [K, VELVET] mode
5 "deep":  int [0,1] deep
```

Double Vocal

```
0 "mdl":   DOUBLE
1 "mode": str [TIGHT, LOOSE, GROUP,
               DETUNE, THICK] mode
2 "mix":  linf [0, 100, 101] %, mix
3 "sprd": linf [0, 100, 101] %, spread
```

Pitch Fix

```
0 "mdl":   PCORR
1 "spd":   linf [1, 100, 100] speed
2 "amnt":  linf [0, 50, 51] amount
3 "a4":    linf [410, 470, 601] A4 pitch
4 "_c":    int [0, 1]
5 "_db":   int [0, 1]
6 "_d":    int [0, 1]
7 "_eb":   int [0, 1]
8 "_e":    int [0, 1]
9 "_f":    int [0, 1]
10 "_gb":  int [0, 1]
11 "_g":   int [0, 1]
12 "_ab":  int [0, 1]
13 "_a":   int [0, 1]
14 "_bb":  int [0, 1]
15 "_b":   int [0, 1]
```

Rotary Speaker

```
0 "mdl":   ROTARY
1 "sw":    str [STOP, SLOW, FAST]
2 "lo":    logf [.1, 3.999, 51] Hz, lo speed
3 "hi":    logf [4, 10, 51] Hz, hi speed
4 "bal":   linf [-100, 100, 201] balance
5 "mix":   linf [0, 100, 101] %, mix
6 "dist":  linf [0, 100, 101] distance
7 "dac":   linf [0, 100, 101] %, drum accel
8 "hac":   linf [0, 100, 101] %, horn accel
```

Phaser

```
0 "mdl":    PHASER
1 "spd":    logf [.05, 5, 201] Hz, speed
2 "phase":  int [0…180] phase
3 "wave":   int [-50…50] wave
4 "range":  int [2…98] %, range
5 "depth":  int [0…100] %, depth
6 "emod":   int [-100, 100] % env mod
7 "att":    logf [10, 1000, 201] ms, attack
8 "hld":    logf [10, 2000, 201] ms, hold
```

	9 “rel”: logf [10, 1000, 201] ms, release 10 “mix”: int [0…100] %, mix 11 “stg”: int [2…12] stages 12 “reso”: int [0…80] %, reso
	Tremolo Panner 0 “mdl”: PANNER 1 “att”: logf [10, 1000, 201] ms, attack 2 “hld”: logf [10, 2000, 201] ms, hold 3 “rel”: logf [10, 1000, 201] ms, release 4 “espd”: int [0…100] %, env>depth 5 “edep”: int [0…100] %, env>depth 6 “spd”: logf [.05, 5, 201] Hz, speed 7 “phase”: int [0…180] phase 8 “wave”: int [-50…50] wave 9 “depth”: int [0…100] %, depth
	Tape Machine 0 “mdl”: TAPE 1 “drv”: linf [-12, 12, 97] dB, drive 2 “spd”: logf [7.5, 30, 65] 3 “low”: int [0, 1] low bump 4 “hi”: int [0, 1] high shelv 5 “out”: linf [-12, 12, 97] dB, out gains s
	Mood Filter 0 “mdl”: MOOD 1 “fbase”: logf [20, 15000, 101] Hz, base 2 “filt”: str [LP, HP, BP, NOTCH] type 3 “slope”: str [12, 24] slope 4 “reso”: linf [0, 10, 101] reso 5 “drv”: linf [0, 10, 101] drive 6 “env”: linf [-100, 100, 201] %, env 7 “att”: logf [10, 250, 101] ms, attack 8 “hld”: logf [1, 500, 101] ms, hold 0 “rel”: logf [1, 500, 101] ms, release 1 “mix”: linf [0, 10, 101] %, mix 2 “lfo”: linf [linf [0, 10, 101] %, lfo 6 “spd”: logf [.05, 20, 301] Hz, speed 7 “phase”: int [0…180] phase 8 “wave”: str [TRI, SIN, SAW+, SAW-, RMP, SQU, RND] lfo wave
	Bodyrez 0 “mdl”: BODY 1 “body”: linf [0, 100, 101] body
	Rack Amp 0 “mdl”: RACKAMP 1 “pre”: linf [0, 10, 101] preamp 2 “buzz”: linf [0, 10, 101] buzz 3 “punch”: linf [0, 10, 101] punch 4 “crunch”:linf [0, 10, 101] crunch 5 “drive linf [0, 10, 101] drive 6 “out”: linf [0, 10, 101] out gain 7 “leq”: linf [0, 10, 101] low eq 8 “heq” linf [0, 10, 101] high eq 9 “cab”: int [0, 1] cab sim

UK Rock Amp

```
 0 "mdl":   UKROCK
 1 "gain":  linf [0, 10, 101] gains
 2 "bass":  linf [0, 10, 101] bass
 3 "mid":   linf [0, 10, 101] middle
 4 "treb":  linf [0, 10, 101] treble
 5 "pres    linf [0, 10, 101] presence
 6 "mstr":  linf [0, 10, 101] master
 7 "out":   linf [0, 10, 101] out gain
 8 "sag"    linf [0, 10, 101] sag
 9 "cab":   int [0, 1] cab sim
```

Angel Amp

```
 0 "mdl":   ANGEL
 1 "gain":  linf [0, 10, 101] gains
 2 "bass":  linf [0, 10, 101] bass
 3 "mid":   linf [0, 10, 101] middle
 4 "treb":  linf [0, 10, 101] treble
 5 "pres    linf [0, 10, 101] presence
 6 "mstr":  linf [0, 10, 101] master
 7 "out":   linf [0, 10, 101] out gain
 8 "sag"    linf [0, 10, 101] sag
 9 "cab":   int [0, 1] cab sim
10 "midb":  int [0, 1] mid boost
11 "bri":   int [0, 1] bright
12 "bt":    int [0, 1] bottom
```

Jazz Clean Amp

```
 0 "mdl":   JAZZC
 1 "vol":   linf [0, 10, 101] volume
 2 "bass":  linf [0, 10, 101] bass
 3 "mid":   linf [0, 10, 101] middle
 4 "treb":  linf [0, 10, 101] treble
 5 "out":   linf [0, 10, 101] out gain
 6 "bri":   int [0, 1] bright
 7 "cab":   int [0, 1] cab sim
```

Deluxe Amp

```
 0 "mdl":   DELUXE
 1 "vol":   linf [1, 10, 91] volume
 2 "bass":  linf [1, 10, 91] bass
 4 "treb":  linf [1, 10, 91] treble
 5 "out":   linf [1, 10, 91] out gain
 6 "sag":   linf [1, 10, 91] sag
 7 "cab":   int [0, 1] cab sim
```

Soul Analogue

```
 0 "mdl":   SOUL
 1 "mix":   linf [0, 125, 126] %, mix
 2 "lf":    linf [0, 10, 101] lo freq
 3 "lg":    linf [-5, 5, 101] lo gain
 4 "lmf":   linf [0, 10, 101] lm freq
 5 "lmf3":  int [0, 1] lm /3
 6 "lmq":   linf [0, 10, 101] lm q
 7 "lmg":   linf [-5, 5, 101] lm gain
 8 "hmf":   linf [0, 10, 101] hm freq
 9 "hmf3":  int [0, 1] hm x3
10 "hmq":   linf [0, 10, 101] hm q
11 "hmg":   linf [-5, 5, 101] hm gain
12 "hf":    linf [0, 10, 101] hf freq
13 "hg":    linf [-5, 5, 101] hf gain
```

Even 88 Formant

```
 0 "mdl":   E88
 1 "mix":   linf [0, 125, 126] %, mix
 2 "lf":    linf [0, 10, 101] lf freq
 3 "lg":    linf [-5, 5, 101] lf gain
 4 "lq":    str [LOW, HIGH] lf q
 5 "lt":    str [BELL, SHELV] lf type
 6 "lmf":   linf [0, 10, 101] lm freq
 7 "lmg":   linf [-5, 5, 101] lm gain
 8 "lmq":   linf [0, 10, 101] lm q
 9 "hmf":   linf [0, 10, 101] hm freq
10 "hmg":   linf [-5, 5, 101] hm gain
11 "hmq":   linf [0, 10, 101] hm q
12 "hf":    linf [0, 10, 101] hm freq
13 "hg":    linf [-5, 5, 101] hf gain
14 "hq":    str [LOW, HIG] hf q
15 "ht":    str [BELL, SHELV] hf type
```

Even 84

```
 0 "mdl":   E84
 1 "mix":   linf [0, 125, 126] %, mix
 2 "g":     linf [-20, 20, 81] dB, gain
 3 "lf":    str [OFF, 35, 60, 110, 220] lf freq
 4 "lg":    linf [-5, 5, 101] lf gain
 5 "mf":    str [OFF, 350, 700, 1k6, 3k2,
                 4k8, 7k2] mid freq
 6 "mg":    linf [-5, 5, 101] mid gain
 7 "mq":    str [LOW, HIGH] mid q
 8 "hf":    str [10k, 12k, 16k, OFF] hf freq
 9 "hg":    linf [-5, 5, 101] hf gain
```

Fortissimo110

```
 0 "mdl":   F110
 1 "mix":   linf [0, 125, 126] %, mix
 2 "peq":   int [0, 1] peq on
 3 "lmf":   linf [0, 10, 101] lm freq
 4 "lmg":   linf [-5, 5, 101] lm gain
 5 "lmq":   linf [0, 10, 101] lm q
 6 "lmf3":  int [0, 1] lm /3
 7 "hmf":   linf [0, 10, 101] hm freq
 8 "hmg":   linf [-5, 5, 101] hm gain
 9 "hmq":   linf [0, 10, 101] hm q
10 "hmf3":  int [0, 1] hm x3
11 "shv":   inf [0, 1] shv on
12 "lf":    str [33, 56, 95, 160,
                 270, 460] lf freq
13 "lg":    linf [-5, 5, 101] lf gain
14 "hf":    str [3k3, 4k7, 6k8, 10k,
                 15k, 18k] hf freq
15 "hg":    linf [-5, 5, 101] hf q
16 "g":     linf [-18, 18, 73] gain
```

Pulsar

```
 0 "mdl":   PULSAR
 1 "mix":   linf [0, 125, 126] %, mix
 2 "eq1":   int [0, 1] eq1 on
 3 "1lb":   linf [0, 10, 101] lf boost
 4 "1latt": linf [0, 10, 101] lf att
 5 "1lf":   str [20, 30, 60, 100] Hz, lf freq
 6 "1hw":   linf [0, 10, 101] hf wid
 7 "1hb":   linf [0, 10, 101] hf boost
 8 "1hf":   str [3k, 4k, 5k, 8k, 10k,
                 12k, 16k] Hz, hf freq
 9 "1hatt": linf [0, 10, 101] hf att
10 "1hattf":str [5k, 10k, 20k] hf att
11 "eq5":   inf [0, 1] eq5 on
12 "5lb":   linf [0, 10, 101] lm boost
```

	13 **"5lf"**: str [200, 300, 500, 700, 1k] Hz, lf freq 14 **"5md"**: linf [0, 10, 101] mid dip 15 **"5mf"**: str [200, 300, 500, 700, 1k, 1k5, 2k, 3k, 4k, 5k, 7k] Hz, mid freq 16 **"5hb"**: linf [0, 10, 101] HM boost 17 **"5hf"**: str [1k5, 2k, 3k, 4k, 5k] Hz, hf freq
	Mach EQ4 0 **"mdl"**: MACH4 1 **"mix"**: linf [0, 125, 126] %, mix 2 **"sub"**: linf [-5, 5, 101] sub 3 **"40"**: linf [-5, 5, 101] 40 4 **"160"**: linf [-5, 5, 101] 160 5 **"650"**: linf [-5, 5, 101] 650 6 **"2k5"**: linf [-5, 5, 101] 2k5 7 **"air"**: linf [0, 10, 101] air 8 **"airm"**: str [OFF, 2k5, 5k, 10k, 20k, 40k] air mode 9 **"again"**: int [0, 1] auto

Premium effects

	None *0 **"mdl"**: NONE*
	*External* *0 **"mdl"**: EXT* *1 **"egrp"**: str [OFF, LCL, AUX, A, B, C, SC, USB, CRD, MOD, PLAY, AES] ext grp* *2 **"ein"**: int [1…64] ext in* *3 **"emode"**: str [M, ST, M/S] ext mode* *4 **"lat"**: int [0…200] latency* *5 **"trim"**: linf [-18, 18, 361] dB, trim*
	Hall Reverb *0 **"mdl"**: HALL* *1 **"pdel"**: int [0…200] ms, pre-delay* *2 **"size"**: int [0…100] hall size* *3 **"dcy"**: logf [.2, 5, 101] s, decay* *4 **"mult"**: logf [.5, , 101] bass multiplier* *5 **"damp"**: logf [1k, 20k, 51] Hz, damping* *6 **"lc"**: logf [20, 400, 51] Hz, low cut* *7 **"hc"**: logf [200, 20k, 51] Hz, high cut* *8 **"shp"**: linf [0, 50, 51] shape* *9 **"sprd"**: int [0…50] spread* *10 **"diff"**: int [1…30] diffusion* *11 **"mspd"**: int [0…100] mod speed*
	Room Reverb *0 **"mdl"**: R-OOM* *1 **"pdel"**: int [0…200] ms, pre-delay* *2 **"size"**: linf [4, 76, 145] m, room size* *3 **"dcy"**: logf [.3, 25, 101] s, decay* *4 **"mult"**: logf [.25, 4, 101] bass multiplier* *5 **"damp"**: logf [1k, 20k, 51] Hz, damping* *6 **"lc"**: logf [20, 400, 51] Hz, low cut* *7 **"hc"**: logf [200, 20k, 51] Hz, high cut* *8 **"shp"**: linf [0, 250, 51] shape* *9 **"sprd"**: int [0…50] spread* *10 **"diff"**: int [0…100] diffusion* *11 **"spin"**: int [0…100] spin* *12 **"ecl"**: linf [0, 1200, 1201] ms, echo left* *13 **"ecr"**: linf [0, 1200, 1201] ms, echo right* *14 **"efl"**: linf [-100, 100, 201] %, feed left* *15 **"efr"**: linf [-100, 100, 201] %, feed right*
	Chamber Reverb *0 **"mdl"**: CHAMBER* *1 **"pdel"**: int [0…200] ms, pre-delay* *2 **"size"**: linf [4, 76, 145] m, room size* *3 **"dcy"**: logf [.3, 25, 101] s, decay* *4 **"mult"**: logf [.25, 4, 101] bass multiplier* *5 **"damp"**: logf [1k, 20k, 51] Hz, damping* *6 **"lc"**: logf [20, 400, 51] Hz, low cut* *7 **"hc"**: logf [200, 20k, 51] Hz, high cut* *8 **"shp"**: linf [0, 250, 51] shape* *9 **"sprd"**: int [0…50] spread* *10 **"diff"**: int [0…100] diffusion* *11 **"spin"**: int [0…100] spin* *12 **"ecl"**: linf [0, 300, 301] ms, echo left* *13 **"ecr"**: linf [0, 300, 301] ms, echo right* *14 **"ell"**: fader lvl dB, echo left* *15 **"elr"**: fader lvl dB, echo right*

Plate Reverb

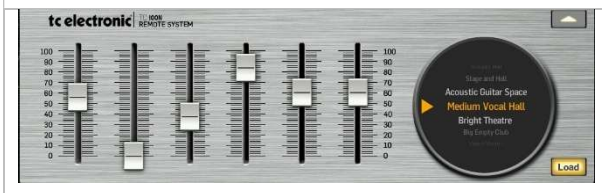
```
 0 "mdl":  PLATE
 1 "pdel": int [0…200] ms, pre-delay
 2 "size": linf [4, 76, 145] m, room size
 3 "dcy":  logf [.3, 25, 101] s, decay
 4 "mult": logf [.25, 4, 101] bass multiplier
 5 "damp": logf [1k, 20k, 51] Hz, damping
 6 "lc":   logf [20, 400, 51] Hz, low cut
 7 "hc":   logf [200, 20k, 51] Hz, high cut
 8 "att":  linf [0, 100, 101] attack
 9 "sprd": int [0…50] spread
10 "diff": int [0…100] diffusion
11 "spin": int [0…100] spin
12 "ecl":  linf [0, 1200, 1201] ms, echo left
13 "ecr":  linf [0, 1200, 1201] ms, echo right
14 "efl":  linf [-100, 100, 201] %, feed left
15 "efr":  linf [-100, 100, 201] %, feed right
```

Concert Reverb

```
 0 "mdl":  CONCERT
 1 "pdel": int [0…200] ms, pre-delay
 2 "size": linf [20, 76, 113] m, room size
 3 "dcy":  logf [.3, 29, 51] s, decay
 4 "mult": logf [.25, 4, 101] bass multiplier
 5 "damp": logf [1k, 20k, 51] Hz, damping
 6 "lc":   logf [20, 400, 51] Hz, low cut
 7 "hc":   logf [200, 20k, 51] Hz, high cut
 8 "shp":  linf [0, 50, 51] shape
 9 "sprd": int [0…50] spread
10 "diff": int [1…16] diffusion
11 "depth":int [0, 100] depth
12 "rfl":  linf [0, 1200, 1201] ms, refl. left
13 "rfr":  linf [0, 1200, 1201] ms, refl. right
14 "rfll": fader lvl dB, reflection left
15 "rflr": fader lvl dB, reflection right
16 "spin": int [0…100] spin
17 "crs":  int [1…100] chorus
```

Ambiance

```
 0 "mdl":  AMBI
 1 "pdel": int [0…200] ms, pre-delay
 2 "size": linf [2, 100, 99] m, room size
 3 "dcy":  logf [.2, 7.3, 101] s, decay
 4 "tail": int [0…100] tail gain
 5 "damp": logf [1k, 20k, 51] Hz, damping
 6 "diff": int [1…30] diffusion
 7 "mod":  int [1…100] modulation speed
 8 "lc":   logf [20, 400, 51] Hz, low cut
 9 "hc":   logf [200, 20k, 51] Hz, high cut
```



VSS3 Reverb

```
 0 "mdl":    VSS3
 1 "preset": str [Build, Small Booth, Home Room,
                  Dialog Alley, Small Wood Room,
                  A Small Room, Intimate Studio,
                  Small Room, Tight & Natural, Room
                  Conversation, Furnished Room 2,
                  Studio 20x20 ft, Drew Room, Piano
                  Close, Clear Guitar Room, Wide
                  Ambient Chamber, Small Dense Hall,
                  Slap Hall, Acoustic Gtr Ambience,
                  Clear Room, Livingroom, Band
                  Rehearsal Room, The Studio, In The
                  Room, Studio 40x40 ft, Hit Room,
                  Ambient Hall, Stage and Hall,
                  Acoustic Guitar Space, Medium
                  Vocal Hall, Bright Theatre , Big
```

```
                    Empty Club, Venue Warm 1, Concert
                    1, Bright Guitar Hall, Concert
                    Arena, Concert Piano, Piano Hall
                    1st Row, Empty Arena, Ballad Vocal
                    Hall, Grand Vocal Hall, Large Warm
                    Hall, Back There, WoodHall,
                    Church, Sound Col, 5000 Hall,
                    Cathedral, Large Church, Medium
                    Church, Warm Cathedral, Cologne
                    Cathedral, Drum Plate Stuff, Drum
                    Wood Plate, Piano Plate, Stairway
                    Plate, Slapback Plate, Ambient
                    Plate, Silky Gold Plate, Gold
                    Plate, EMT 141, Leader Of The
                    Band, VocalDry, Vocal Room,
                    VocalWet, Slapback Vox 1, Vocal
                    Hall 1, Vocal Chamber, Bright Male
                    Vox, Vocal Bright, Vocal Deep,
                    Vocal Female, Vocal Deep Male,
                    Large Vocal Hall, Kick & Bass
                    Ambience, Drum Room, Small Perc
                    Room, Drum Room Xpander, Bright
                    Shoe Gaze Snare, Snare Room
                    Bright, Tom-Tom Reverb, Bossa Nova
                    Perc Room, Hard Drum Space, Puk
                    Drum Ambience, Overhead Mics,
                    Dance Snare, Drum Perc Soft 1,
                    Perc Straight Tail, Store Room,
                    Studio Small, The Alley, Near The
                    Wall, WoodFlr, Large Office,
                    Conference Room, Dance Studio,
                    Forest 2, StoneWall, Venue 1,
                    Small Stairway, Forest 1, Airport
                    PA, Small Tower Hall, On The
                    Street, Dark Tunnel, Empty
                    Nightclub, Parking Garage, Parking
                    Distant, Long Swimmingpool] preset
 2 "load":   int [0, 1] load
 3 "erpdly": linf [0, 100, 101] ms, er pdly
 4 "ertype": str [ROYAL, THEATRE, CHURCH, GAS,
                    CONCERT, ROYAL2, V1-NEAR,
                    V2-HARD, V3-SPREAD, V4-BUILD,
                    V5-RANDOM, SLAP, CAR, PHONEBTH,
                    BATHROOM, CONFRM9, CONFRM30,
                    GARAGE, SWIMSTDM, AIRPORT,
                    STREET, ALLEY, PIAZA,
                    FOREST], er type
 5 "ersize": str [SML, MED, LRG] er size
 6 "erpos":  str [NEAR, DIST] er position
 7 "erbal":  linf [-100, 100, 201], er balance
 8 "erlc":   logf [20, 400, 51] Hz, er low cut
 9 "ercol":  linf [-40, 40, 81] er color
10 "erlvl":  fader lvl dB, er level
11 "rvtype": str [SMOOTH, NATURAL, ALIVE, FAST,
                    X-WIDE, ALIVE2] rev type
12 "rvwide": str [NARROW, NORMAL, WIDE, X-WIDE] rev
                    wide
13 "rvpdly": linf [0, 200, 201] ms, rev pdly
14 "dcy":    linf [.1, 20, 280] s, decay
15 "diff":   linf [-50, 50, 101] diffuse
16 "rvbal":  linf [-100, 100, 201] balance
17 "rvlvl":  fader lvl dB, reverb level
18 "ldcy":   linf [.1, 2.5, 25] low decay
19 "lmdcy":  linf [.1, 2.5, 25] lowmid decay
20 "hmdcy":  linf [.1, 2.5, 25] mid decay
21 "hdcy":   linf [.1, 2.5, 25] high decay
22 "hsoft":  linf [-50, 50, 101] high soft
23 "lxo":    logf [20, 500, 113] Hz, low xover
24 "mxo":    logf [200, 2000, 81] Hz, mid xover
25 "hxo":    logf [500, 20000, 105] Hz, high xover
```

	26 **"lshv"**: logf [20, 200, 81] Hz, low shelf 27 **"lsdmp"**: linf [0, -18, 37] dB, low damp 28 **"hcut"**: logf [20, 20000, 241] Hz, high cut 29 **"mtype"**: str [A, B, C, D, E, F] modulation type 30 **"mrate"**: linf [-100, 100, 201] modulation rate 31 **"mwid"**: linf [0, 200, 201] modulation width 32 **"view"**: int [0, 1] view
	Vintage Room 0 **"mdl"**: V-ROOM 1 **"pdel"**: int [0…200] ms, pre-delay 2 **"size"**: int [0…50] size 3 **"dcy"**: logf [.1, 20, 101] s, decay 4 **"dens"**: linf [1, 30, 30] density 5 **"erlvl"**: linf [0, 100, 101] %, Early level 6 **"lmult"**: logf [.1, 10, 101] low multiplier 7 **"hmult"**: logf [.1, 10, 101] high multiplier 8 **"lc"**: logf [20, 400, 51] Hz, low cut 9 **"hc"**: logf [200, 20k, 51] Hz, high cut 10 **"frz"**: int [0, 1] freeze 11 **"erl"**: linf [0, 200, 201] ms, er delay left 12 **"err"**: linf [0, 200, 201] ms, er delay right 13 **"add"**: int [0, 1] add 14 **"lvl"**: int [-6, 6, 101] dB, level
	Vintage Reverb, 0 **"mdl"**: V-REV 1 **"pdel"**: int [0…120] ms, pre-delay 2 **"dcy"**: linf [.4, 4.5, 83] s, decay 3 **"lmult"**: logf [.5, 2, 51] low multiplier 4 **"hmult"**: logf [.25, .67, 51] high multiplier 5 **"mod"**: int [0…100] modulation speed 6 **"lc"**: logf [20, 400, 51] Hz, low cut 7 **"hc"**: logf [5000, 20k, 31] Hz, high cut 8 **"out"**: str [FRONT, REAR] output 9 **"trans"**: int [0…1] transformer
	Vintage Plate 0 **"mdl"**: V-PLATE 1 **"pdel"**: int [0…250] ms, pre-delay 2 **"dcy"**: linf [1, 6, 101] s, decay 3 **"lc"**: logf [20, 400, 51] Hz, low cut 4 **"col"**: linf [-20, 20, 42] color
	Blue Plate 0 **"mdl"**: BPLATE 1 **"pdel"**: int [0…200] ms, pre delay 2 **"size"**: int [0…100] ms, size 3 **"dcy"**: logf [0.2, 5, 101] s, decay 4 **"mult"**: logf [0.5, 2, 51] bass multiplier 5 **"damp"**: logf [1000, 20000, 51] Hz, damping 6 **"lc"**: logf [20, 400, 51] Hz, low cut 7 **"hc"**: logf [200, 20000, 51] Hz, high cut 8 **"xover"**: logf [20, 500, 51] Hz, xover 9 **"mdep"**: linf [1, 50, 50] modulation depth 10 **"msdp"**: int [0…100] modulation speed 11 **"diff"**: int [1…30] diffusion
	Gated Reverb 0 **"mdl"**: GATED 1 **"pdel"**: int [0…200] ms, pre-delay 2 **"att"**: int [4…30] attack 3 **"dcy"**: logf [.14, 1, 101] s, decay 4 **"dens"**: int [0…100] density 5 **"diff"**: int [0…100] diffusion

	6 **“sprd”**: int [0…50] spread 7 **“lc”**: logf [20, 400, 51] Hz, low cut 8 **“hfs”**: logf [200, 20k, 51] Hz, high freq 9 **“hsg”**: linf [-30, 0, 61] dB, high gain
	Reverse Reverb 0 **“mdl”**: REVERSE 1 **“pdel”**: int [0…200] ms, pre-delay 2 **“rise”**: int [4…50] rise 3 **“dcy”**: logf [.14, 1, 101] s, decay 4 **“diff”**: int [0…30] diffusion 5 **“sprd”**: int [0…100] spread 6 **“lc”**: logf [20, 400, 51] Hz, low cut 7 **“hfs”**: logf [200, 20k, 51] Hz, high freq 8 **“hsg”**: linf [-30, 0, 61] dB, high gain
	Delay/Reverb 0 **“mdl”**: DEL/REV 1 **“time”**: linf [0, 3000, 3000] ms, time 2 **“feed”**: linf [0, 100, 101] %, feed 3 **“fhc”**: logf [200, 2000, 51] Hz, feed HC 4 **“dly”**: linf [0, 100, 101] %, delay 5 **“d2r”**: linf [0, 100, 101] %, delay→rev 6 **“pdel”**: int [0…200] ms, pre delay 7 **“size”**: int [2…100] size 8 **“dcy”**: logf [.1, 5, 51] s, decay 9 **“damp”**: logf [1000, 20k, 51] Hz, damp 10 **“rlc”**: logf [20, 400, 51] Hz, rev LC 11 **“i2r”**: linf [0, 100, 101] %, in→rev
	Shimmer Reverb 0 **“mdl”**: SHIMMER 1 **“pdel”**: int [0…250] ms, pre delay 2 **“size”**: int [2…50] size 3 **“dcy”**: logf [1, 20, 101] s, decay 4 **“lc”**: logf [25, 250, 51] Hz, low cut 5 **“hc”**: logf [500, 7000, 51] Hz, high cut 6 **“damp”**: linf [0, 100, 101] %, damp 7 **“shim”**: linf [0, 100, 101] %, shimmer 8 **“shine”**: linf [0, 100, 101] %, shine
	Spring Reverb 0 **“mdl”**: SPRING 1 **“dcy”**: logf [1.5, 6, 101] s, decay 2 **“dens”**: linf [1, 30, 30] density 3 **“low”**: linf [1, 50, 50] bass 4 **“high”**: linf [1, 50, 50] treble
	Dimension CRS 0 **“mdl”**: DIMCRS 1 **“sw1”**: int [0, 1] sw1 2 **“sw2”**: int [0, 1] sw2 3 **“sw3”**: int [0, 1] sw3 4 **“sw4”**: int [0, 1] sw4 5 **“in”**: str [MONO, STEREO] input 6 **“drysw”**: int [0, 1] dry
	Stereo Chorus 0 **“mdl”**: CHORUS 1 **“lc”**: logf [20, 400, 51] Hz, LC 2 **“hc”**: logf [200, 20000, 51] Hz, HC 3 **“wave”**: linf [0, 100, 101] waveform 4 **“phase”**: linf [0, 100, 101] phase 5 **“mix”**: linf [0, 100, 101] %, mix

	6 **“dlyl”**: linf [5, 50, 226] ms, delay l 7 **“dlyr”**: linf [5, 50, 226] ms, delay r 8 **“depl”**: linf [0, 100, 101] %, depth l 9 **“depr”**: linf [0, 100, 101] %, depth r 10 **“sprd”**: linf [0, 100, 101] %, spread 11 **“spd”**: logf [.05, 5, 201] Hz, speed
	Stereo Flanger 0 **“mdl”**: CHORUS 1 **“lc”**: logf [20, 400, 51] Hz, LC 2 **“hc”**: logf [200, 20000, 51] Hz, HC 3 **“flc”**: logf [20, 400, 51] Hz, feed LC 4 **“fhc”**: logf [200, 20000, 51] Hz, feed HC 5 **“mix”**: linf [0, 100, 101] %, mix 6 **“dlyl”**: linf [5, 20, 196] ms, delay l 7 **“dlyr”**: linf [5, 20, 196] ms, delay r 8 **“depl”**: linf [0, 100, 101] %, depth l 9 **“depr”**: linf [0, 100, 101] %, depth r 10 **“phase”**: linf [0, 180, 181] phase 11 **“spd”**: logf [.05, 5, 201] Hz, speed 12 **“feed”**: linf [-90, 90, 181] %, feed
	Stereo Delay 0 **“mdl”**: ST-DL 1 **“time”**: linf [1, 3000, 3000] ms, time 2 **“mode”**: str [ST, X, M] mode 3 **“fact”**: str [1/4, 1/3, 1/2, 2/3, 3/4, 1, 3/8, 5/4, 4/3, 3/2, 2] factor 4 **“pat”**: str [1/2:1, 2/3:1, 3/4:1, 7/8:1, 1:1, 1:9/8, 1:5/4, 1:4/3, 1:3/2] pattern 5 **“offset”**:int [-50…50] ms, offset 6 **“feed”**: linf [0, 100, 101] %, feed 7 **“flc”**: logf [20, 400, 51] Hz, feed L cut 8 **“fhc”**: logf [200, 20000, 51] Hz, feed H cut 9 **“lc”**: logf [20, 400, 51] Hz, low cut 10 **“hc”**: logf [200, 20000, 51] Hz, high cut
	UltraTap Delay 0 **“mdl”**: TAP-DL 1 **“time”**: linf [1, 2000, 2000] ms, time 2 **“rep”**: int [1..16] repeat 3 **“slp”**: linf [-6, 6, 121] dB, slope 4 **“fact”**: str [1/3, 1/2, 2/3, 3/4, 1, 5/4, 4/3, 3/2, 2] factor 5 **“pdel”**: linf [0, 500, 501] ms, pre delay 6 **“mode”**: str [MOVE, JUMP, FOCUS, SPREAD] mode 7 **“wid”**: linf [-100, 100, 201] %, width 8 **“diff”**: linf [0, 100, 101] diffusion 9 **“lc”**: logf [20, 400, 51] Hz, low cut 10 **“hc”**: logf [200, 20000, 51] Hz, high cut
	Tape Delay 0 **“mdl”**: TAPE-DL 1 **“time”**: linf [60, 650, 591] ms, time 2 **“sust”**: linf [0, 100, 101] %, sustain 3 **“drv”**: linf [0, 100, 101] %, drive 4 **“wf”**: linf [0, 100, 101] %, flutter

OilCan Delay

```
0 "mdl":   OILCAN
1 "time":  linf [0, 10, 1001] time
2 "sust":  linf [0, 10, 101] %, sustain
3 "wb":    linf [0, 10, 101] %, wobble
4 "tone":  linf [0, 10, 101] %, tone
```

BBD Delay

```
0 "mdl":   BBD-DL
1 "dly":   linf [0, 100, 1001] time
2 "feed":  linf [0, 100, 101] %, feed
```

Stereo Pitch

```
0 "mdl":   PITCH
1 "semi":  int [-12…12] semitones
2 "cent":  int [-50…50] cent
3 "dly":   linf [0, 500, 501] ms, delay
4 "lc":    logf [20, 400, 51] Hz, low cut
5 "hc":    logf [200, 20000, 51] Hz, high cut
6 "mix":   linf [0, 100, 101] %, mix
```

Dual Pitch

```
0 "mdl":   D-PITCH
1 "semi1": int [-12…12] semitones 1
2 "cent1": int [-50…50] cent 1
3 "dly1":  linf [0, 500, 501] ms, delay 1
4 "pan1":  linf [-100, 100, 201] %, pan 1
5 "lvl1":  fader lvl 1 dB
6 "semi2": int [-12…12] semitones 2
7 "cent2": int [-50…50] cent 2
8 "dly2":  linf [0, 500, 501] ms, delay 2
9 "pan2":  linf [-100, 100, 201] %, pan 2
10 "lvl2": fader lvl 2 dB
11 "lc":   logf [20, 400, 51] Hz, low cut
12 "hc":   logf [200, 20000, 51] Hz, high cut
```

Channel effects

	None *0 “mdl”: NONE*
	*External* *0 “mdl”: EXT* *1 “egrp”: str [OFF, LCL, AUX, A, B, C, SC, USB, CRD, MOD, PLAY, AES] ext grp* *2 “ein”: int [1…64] ext in* *3 “emode”: str [M, ST, M/S] ext mode* *4 “lat”: int [0…200] latency* *5 “trim”: linf [-18, 18, 361] dB, trim*
	Soul Analog EQ *0 “mdl”: SOUL* *1 “mix”: linf [0, 125, 126] %, mix* *2 “lf”: linf [0, 10, 101] lo freq* *3 “lg”: linf [-5, 5, 101] lo gain* *4 “lmf”: linf [0, 10, 101] lm freq* *5 “lmf3”: int [0, 1] lm /3* *6 “lmq”: linf [0, 10, 101] lm q* *7 “lmg”: linf [-5, 5, 101] lm gain* *8 “hmf”: linf [0, 10, 101] hm freq* *9 “hmf3”: int [0, 1] hm x3* *10 “hmq”: linf [0, 10, 101] hm q* *11 “hmg”: linf [-5, 5, 101] hm gain* *12 “hf”: linf [0, 10, 101] hf freq* *13 “hg”: linf [-5, 5, 101] hf gain*
	Even 88-Formant EQ *0 “mdl”: E88* *1 “mix”: linf [0, 125, 126] %, mix* *2 “lf”: linf [0, 10, 101] lf freq* *3 “lg”: linf [-5, 5, 101] lf gain* *4 “lq”: str [LOW, HIGH] lf q* *5 “lt”: str [BELL, SHELV] lf type* *6 “lmf”: linf [0, 10, 101] lm freq* *7 “lmg”: linf [-5, 5, 101] lm gain* *8 “lmq”: linf [0, 10, 101] lm q* *9 “hmf”: linf [0, 10, 101] hm freq* *10 “hmg”: linf [-5, 5, 101] hm gain* *11 “hmq”: linf [0, 10, 101] hm q* *12 “hf”: linf [0, 10, 101] hm freq* *13 “hg”: linf [-5, 5, 101] hf gain* *14 “hq”: str [LOW, HIGH] hf q* *15 “ht”: str [BELL, SHELV] hf type*
	Even 84 EQ *0 “mdl”: E84* *1 “mix”: linf [0, 125, 126] %, mix* *2 “g”: linf [-20, 20, 81] dB, gain* *3 “lf”: str [OFF, 35, 60, 110, 220] lf freq* *4 “lg”: linf [-5, 5, 101] lf gain* *5 “mf”: str [OFF, 350, 700, 1k6, 3k2, 4k8, 7k2] mid freq* *6 “mg”: linf [-5, 5, 101] mid gain* *7 “mq”: str [LOW, HIGH] mid q* *8 “hf”: str [10k, 12k, 16k, OFF] hf freq* *9 “hg”: linf [-5, 5, 101] hf gain*

Focusrite ISA 110 EQ

```
 0 “mdl”:    F110
 1 “mix”:    linf [0, 125, 126] %, mix
 2 “peq”:    int [0, 1] peq on
 3 “lmf”:    linf [0, 10, 101] lm freq
 4 “lmg”:    linf [-5, 5, 101] lm gain
 5 “lmq”:    linf [0, 10, 101] lm q
 6 “lmf3”:   int [0, 1] lm /3
 7 “hmf”:    linf [0, 10, 101] hm freq
 8 “hmg”:    linf [-5, 5, 101] hm gain
 9 “hmq”:    linf [0, 10, 101] hm q
10 “hmf3”:   int [0, 1] hm x3
11 “shv”:    inf [0, 1] shv on
12 “lf”:     str [33, 56, 95, 160,
                  270, 460] lf freq
13 “lg”:     linf [-5, 5, 101] lf gain
14 “hf”:     str [3k3, 4k7, 6k8, 10k,
                   15k, 18k] hf freq
15 “hg”:     linf [-5, 5, 101] hf q
16 “g”:      linf [-18, 18, 73] gain
```

Pulsar P1a/M5 EQ

```
 0 “mdl”:    PULSAR
 1 “mix”:    linf [0, 125, 126] %, mix
 2 “eq1”:    int [0, 1] eq1 on
 3 “1lb”:    linf [0, 10, 101] lf boost
 4 “1latt”:  linf [0, 10, 101] lf att
 5 “1lf”:    str [20, 30, 60, 100] Hz, lf freq
 6 “1hw”:    linf [0, 10, 101] hf wid
 7 “1hb”:    linf [0, 10, 101] hf boost
 8 “1hf”:    str [3k, 4k, , 5k, 8k, 10k,
                  12k, 16k] Hz, hf freq
 9 “1hatt”:  linf [0, 10, 101] hf att
10 “1hattf”: str [5k, 10k, 20k] hf att
11 “eq5”:    int [0, 1] eq5 on
12 “5lb”:    linf [0, 10, 101] lm boost
13 “5lf”:    str [200, 300, 500, 700,
                  1k] Hz, lf freq
14 “5md”:    linf [0, 10, 101] mid dip
15 “5mf”:    str [200, 300, 500, 700, 1k, 1k5,
                  2k, 3k, 4k, 5k, 7k] Hz, mid freq
16 “5hb”:    linf [0, 10, 101] HM boost
17 “5hf”:    str [1k5, 2k, 3k, 4k,
                  5k] Hz, hf freq
```

Mach EQ4

```
 0 “mdl”:    MACH4
 1 “mix”:    linf [0, 125, 126] %, mix
 2 “sub”:    linf [-5, 5, 101] sub
 3 “40”:     linf [-5, 5, 101] 40
 4 “160”:    linf [-5, 5, 101] 160
 5 “650”:    linf [-5, 5, 101] 650
 6 “2k5”:    linf [-5, 5, 101] 2k5
 7 “air”:    linf [0, 10, 101] air
 8 “airm”:   str [OFF, 2k5, 5k, 10k,
                  20k, 40k] air mode
 9 “again”: int [0, 1] auto
```

Even Channel

Even 88 Gate, Even 88 Formant EQ, Even Compressor/Limiter

```
 0 “mdl”:     *EVEN*
 1 “g_thr”:   Linf [-40.0, 0.0, 81] dB
 2 “g_hyst”:  Linf [0.0, 25.0, 51] dB
 3 “g_range”: Linf [0, 60, 61] dB
 4 “g_rel”:   Logf [100, 3000, 130] ms
 5 “g_fast”:  int [0, 1]
```

	6 “g_m40”: int [0, 1] 7 “g_on”: int [0, 1] 8 “eq_on”: int [0, 1] 9 “lf”: Linf [0.0, 10.0, 101] 10 “lg”: Linf [-5.0, 5.0, 101] 11 “lq”: Str [LOW, HIGH] 12 “lt”: Str [BELL, SHELV] 13 “lmf”: Linf [0.0, 10.0, 101] 14 “lmg”: Linf [-5.0, 5.0, 101] 15 “lmq”: Linf [0.0, 10.0, 101] 16 “hmf”: Linf [0.0, 10.0, 101] 17 “hmg”: Linf [-5.0, 5.0, 101] 18 “hmq”: Linf [0.0, 10.0, 101] 19 “hf”: Linf [0.0, 10.0, 101] 20 “hg”: Linf [-5.0, 5.0, 101] 21 “hq”: Str [LOW, HIGH] 22 “ht”: Str [BELL, SHELV] 23 “mix”: Linf [0, 125 %, 126] 24 “d_lon”: int [0, 1] 25 “d_lthr”: Linf [-12.0, 0.0, 25] dB 26 “d_lrec”: Str [50, 100, 200, 800, A1, A2] 27 “d_lfast”: int [0, 1] 28 “d_con”: int [0, 1] 29 “d_cthr”: Linf [-35.0, -5.0 dB, 61] 30 “d_ratio”: Str [1.5, 2.0, 3.0, 4.0, 6.0] 31 “d_crec”: Str [100, 400, 800, 1500, A1, A2] 32 “d_cfast”: int [0, 1] 33 “d_gain”: Linf [-6, 12 dB, 7]
	Soul Channel Soul 9000 Gate/Expander, Soul AnaLogue EQ, Soul 9000 Channel Compressor 0 “mdl”: *SOUL* 1 “g_thr”: Linf [-40.0, 0.0, 81] dB 2 “g_range”: Linf [0, 40, 41] dB 3 “g_hld”: Logf [10, 4000, 130] ms 4 “g_rel”: Logf [100, 4000, 130] ms 5 “g_fast”: int [0, 1] 6 “g_mode”: Str [GATE, EXP] 7 “g_on”: int [0, 1] 8 “eq_on”: int [0, 1] 9 “lf”: Linf [0.0, 10.0, 101] 10 “lg”: Linf [-5.0, 5.0, 101] 11 “lmf”: Linf [0.0, 10.0, 101] 12 “lmf3”: int [0, 1] 13 “lmq”: Linf [0.0, 10.0, 101] 14 “lmg”: Linf [-5.0, 5.0, 101] 15 “hmf”: Linf [0.0, 10.0, 101] 16 “hmf3”: int [0, 1] 17 “hmq”: Linf [0.0, 10.0, 101] 18 “hmg”: Linf [-5.0, 5.0, 101] 19 “hf”: Linf [0.0, 10.0, 101] 20 “hg”: Linf [-5.0, 5.0, 101] 21 “mix”: Linf [0, 125 %, 126] 22 “d_on”: int [0, 1] 23 “d_thr”: Linf [-30.0, 18.0, 97] dB 24 “d_ratio”: Str [1.3, 1.4, 1.6, 1.8, 2.0, 2.5, 2.8, 3.3, 4.0, 5.0, 6.0, 7.0, 9.0, 12, 20, 50, 100] 25 “d_fast”: int [0, 1] 26 “d_rel”: Logf [100, 4000, 65] ms 27 “d_peak”: int [0, 1]

Vintage Channel

76 Limiting Amplifier, Pulsar EQ P1A/m5, Model 2A Leveling Amplifier

```
 0 "mdl":      *VINTAGE*
 1 "d_in":     Linf [-48.0, 0.0, 97] dB
 2 "d_out":    Linf [-48.0, 0.0, 97] dB
 3 "d_att":    Linf [1.0, 7.0, 61]
 4 "d_rel":    Linf [1.0, 7.0, 61]
 5 "d_ratio":  Str [4, 8, 12, 20, ALL]
 6 "d_on":     int [0, 1]
 7 "eq1":      int [0, 1]
 8 "1lb":      Linf [0.0, 10.0, 101]
 9 "1latt":    Linf [0.0, 10.0, 101]
10 "1lf":      Str [20, 30, 60, 100]
11 "1hw":      Linf [0.0, 10.0, 101]
12 "1hb":      Linf [0.0, 10.0, 101]
13 "1hf":      Str [3k, 4k, 5k, 8k, 10k, 12k, 16k]
14 "1hatt":    Linf [0.0, 10.0, 101]
15 "1hattf":   Str [5k, 10k, 20k]
16 "eq5":      int [0, 1]
17 "5lb":      Linf [0.0, 10.0, 101]
18 "5lf":      Str [200, 300, 500, 700, 1k]
19 "5md":      Linf [0.0, 10.0, 101]
20 "5mf":      Str [200, 300, 500, 700, 1k, 1k5,
                    2k, 3k, 4k, 5k, 7k]
21 "5hb":      Linf [0.0, 10.0, 101]
22 "5hf":      Str [1k5, 2k, 3k, 4k, 5k]
23 "l_ingain": Linf [0, 100, 101]
24 "l_peak":   Linf [0, 100, 101]
25 "l_mode":   Str [COMP, LIM]
26 "l_on":     int [0, 1]
```

Bus Channel

Soul Warmth, Even 84 EQ, Soul G Bus Compressor

```
 0 "mdl":      *BUS*
 1 "w_drv":    Linf [10, 125, 116] %
 2 "w_hrm":    Linf [-100, 100, 201]
 3 "w_col":    Linf [-1.00, +1.00, 41]
 4 "w_trim":   Linf [-18.0, +6.0, 49] dB
 5 "w_mix":    Linf [0, 100, 101] %
 6 "w_on":     int [0, 1]
 7 "eq_on":    int [0, 1]
 9 "g":        Linf [-20.0, 20.0, 81] dB
10 "lf":       Str [OFF, 35, 60, 110, 220]
11 "lg":       Linf [-5.0, 5.0, 101]
12 "mf":       Str [OFF, 350, 700, 1k6, 3k2,
                    4k8, 7k2]
13 "mg":       Linf [-5.0, 5.0, 101]
14 "mq":       Str [LOW, HIGH]
15 "hf":       Str [10k, 12k, 16k, OFF]
16 "hg":       Linf [-5.0, 5.0, 101]
17 "mix":      Linf [0, 125 %, 126]
18 "d_thr":    Linf [-40.0, 0.0, 81] dB
19 "d_ratio":  Str [1.5, 2.0, 3.0, 4.0, 5.0, 10]
20 "d_att":    Str [0.1, 0.3, 1.0, 3.0, 10.0, 30.0]
21 "d_rel":    Str [0.1, 0.2, 0.4, 0.8, 1.6, AUTO]
22 "d_gain":   Linf [-6.0, 12.0, 37] dB
23 "d_on":     int [0, 1]
```

Mastering

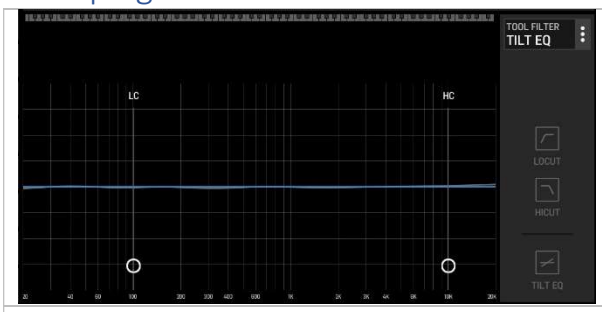
Tape, Mach EQ4 EQ, Stereo Enhancer, Precision Limiter

```
 0 "mdl":      *MASTER*
 1 "t_drv":    Linf [-5.0, 25.0, 61] dB
 2 "t_spd":    Logf [7.5, 30.0, 65]
 3 "t_low":    int [0, 1]
 4 "t_hi":     int [0, 1]
 5 "t_on":     int [0, 1]
```

	6 **“sub”**: Linf [-5.0, 5.0, 201] 7 **“40”**: Linf [-5.0, 5.0, 201] 8 **“160”**: Linf [-5.0, 5.0, 201] 9 **“650”**: Linf [-5.0, 5.0, 201] 10 **“2k5”**: Linf [-5.0, 5.0, 201] 11 **“air”**: Linf [0.0, 10.0, 201] 12 **”airm”**: Str [OFF, 2k5, 5k, 10k, 20k, 40k] 13 **“eq_on”**: int [0, 1] 14 **“e_stlvl”**: Linf [-100, +100, 201] % 15 **“e_lmf”**: Linf [-100, +100, 201] % 16 **“e_mlvl”**: Linf [-100, +100, 201] % 17 **“e_st”**: Linf [-100, 100, 201] % 18 **“e_m”**: Linf [-100, 100, 201] % 19 **“e_bass”**: Linf [0, 100, 101] % 20 **“e_mid”**: Linf [0, 100, 101] % 21 **“e_high”**: Linf [0, 100, 101] % 22 **“e_bassf”**: Linf [1, 50, 50] 23 **“e_midq”**: Linf [1, 50, 50] 24 **“e_highf”**: Linf [1, 50, 50] 25 **“e_on”**: int [0, 1] 26 **“l_gin”**: Linf [0.00, 18.00, 73] dB 27 **“l_gout”**: Linf [-18.00, 0.00, 73] dB 28 **“l_sqz”**: int [0, 100] 29 **“l_knee”**: int [0, 10] 30 **“l_again”**: int [0, 1] 31 **“l_att”**: Linf [0.05, 1.00, 96] ms 32 **“l_rel”**: Logf [20, 2000, 101] ms 33 **“l_on”**: int [0, 1]

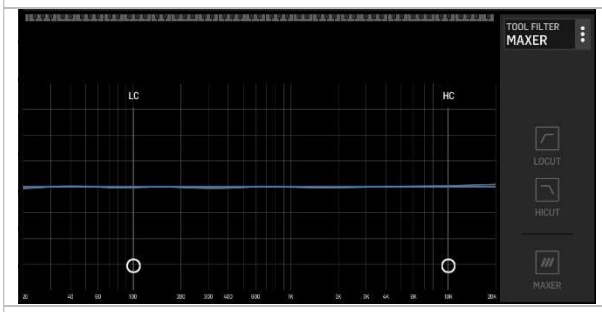
Plugins

Filter plugins



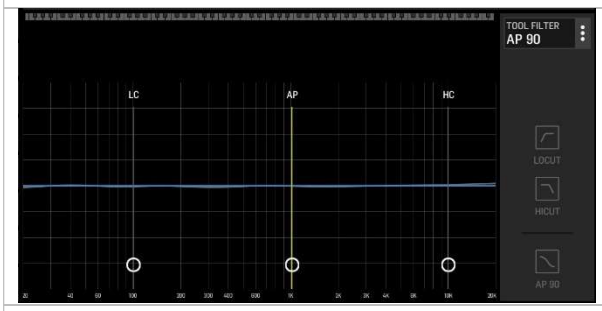
Tilt Filter

```
0 "mdl":   TILT
1 "tilt":  linf [-6, 6, 49] tilt
```



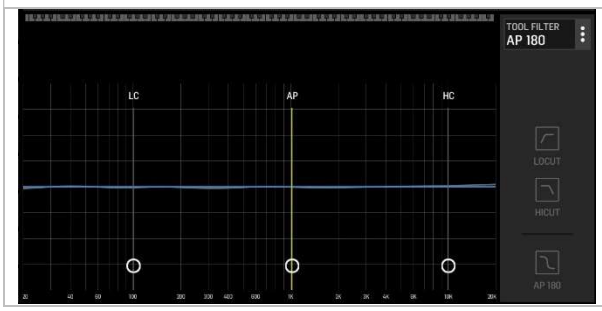
Maxer Filter

```
0 "mdl":   MAX
1 "low":   linf [0, 100, 101] %, low cont
2 "proc":  linf [0, 100, 101] %, high proc
```



AP90 Filter (all pass)

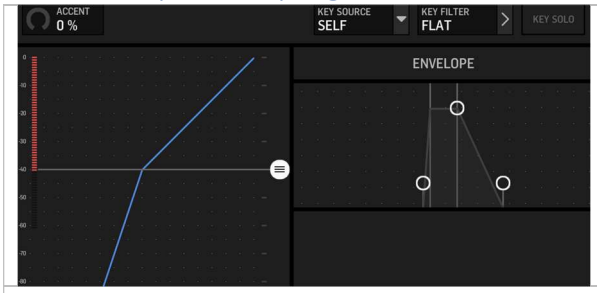
```
0 "mdl":   AP1
1 "freq":  logf [100, 10000, 100] Hz, freq
```



AP180 Filter (all pass)

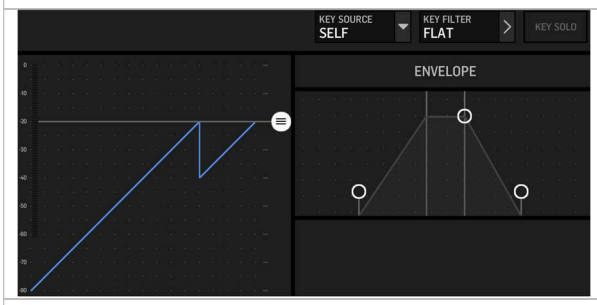
```
0 "mdl":   AP2
1 "f":     logf [100, 10000, 100] Hz, freq
2 "q":     logf [.442, 10, 181] q
```

Gate/Compressor plugins



Standard Gate/Expander

```
0 "mdl":    GATE
1 "thr":    linf [-80, 0, 161] dB, thr
2 "range":  linf [3, 60, 115] dB, range
3 "att":    linf [0, 120, 121] ms, attack
4 "hld":    linf [1, 1000, 121] ms, hold
5 "rel":    logf [4, 4000, 121] ms, release
6 "acc":    linf [0, 100, 21] %, accent
7 "ratio":  str [1:1.5, 1:2, 1:3, 1:4,
                 gate] ratio
```



Standard Ducker

```
0 "mdl":    DUCK
1 "thr":    linf [-80, 0, 161] dB, thr
2 "range":  linf [3, 60, 115] dB, range
3 "att":    linf [0, 120, 121] ms, attack
4 "hld":    linf [1, 4000, 121] ms, hold
5 "rel":    linf [20, 4000, 121] ms, release
```

Even 88-Gate

```
0 "mdl":    E88
1 "thr":    linf [-40, 0, 81] dB, thr
2 "hyst":   linf [0, 25, 51] dB, hyst
3 "range":  linf [0, 60, 61] dB, range
4 "rel":    logf [100, 3000, 130] ms, release
5 "fast":   int [0, 1] fast
6 "m40":    int [0, 1] thr
```

SSL 9000 Gate/Expander

```
0 "mdl":     9000G
1 "thr":     linf [-40, 0 81] dB, input
2 "range":   linf [-0, 40, 41] dB
3 "hld":     logf [10, 4000, 130] ms, hold
4 "rel":     logf [100, 4000, 130] ms, release
5 "fast":    int [0, 1] fast
6 "mode":    str [GATE, EXP] mode
```

DrawMore Expander Gate 241

```
0 "mdl":    D241G
1 "thr":    linf [-80, 0, 161] dB, thr
2 "slow":   int [0, 1] slow
```

DBX 902 De-Esser

```
0 "mdl":    DS902
1 "f":      logf [800, 8000, 130] Hz, freq
2 "range":  linf [3, 12, 25] dB, range
3 "mode":   str [FULL, HF] mode
```

Dynamic EQ

```
 0 “mdl”:   DEQ
 1 “thr”:   linf [-60, 0, 121] dB, thr
 2 “ratio”: flt [1.2, 1.3, 1.5, 2.0,
                 3.0, 5.0, 10.0] ratio
 3 “att”:   linf [0, 200, 201] ms, attack
 4 “rel”:   logf [20, 4000, 130] ms, release
 5 “filt”:  str [OFF, BP, LP6, LP12,
                 HP6, HP12] filter
 6 “g”:     linf [-15, 15, 301] dB, gain
 7 “f”:     logf [20, 20000, 961] Hz, freq
 8 “q”:     logf [.442, 10, 181] q
 9 “mode”:  str [low, high] mode
```

Dual Dynamic EQ

```
 0 “mdl”:    DEQ2
 1 “1-thr”:  linf [-60, 0, 121] dB, threshold 1
 2 “1-ratio”:str [1.20, 1.30, 1.50, 2.00, 3.00,
                  5.00, 10.00] ms, ratio 1
 3 “1-att”:  linf [0.00, 200.00, 201] ms, att 1
 4 “1-rel”:  logf [20.00, 4000.00, 130] ms, rel1
 5 “1-filt”: str [OFF, BP, LP6, LP12, HP6, HP12]
 6 “1-g”:    linf [-15.00, 15.00, 301] dB, gain1
 7 “1-f”:    logf [20, 20000, 961] Hz, freq 1
 8 “1-q”:    logf [0.44, 10.00, 181] qual 1
 9 “1-mode”: str [low, high] mode 1
10 “2-thr”:  linf [-60, 0, 121] dB, threshold 2
11 “2-ratio”:str [1.20, 1.30, 1.50, 2.00, 3.00,
                  5.00, 10.00] ms, ratio 2
12 “2-att”:  linf [0.00, 200.00, 201] ms, att 2
13 “2-rel”:  logf [20.00, 4000.00, 130] ms, rel2
14 “2-filt”: str [OFF, BP, LP6, LP12, HP6, HP12]
15 “2-g”:    linf [-15.00, 15.00, 301] dB, gain2
16 “2-f”:    logf [20, 20000, 961] Hz, freq 2
17 “2-q”:    logf [0.44, 10.00, 181] qual 2
18 “2-mode”: str [low, high] mode 2
19 “$page”:  int [0..1] page
```

Wave Designer

```
 0 “mdl”:   WAVE
 1 “att”:   linf [-15, 15, 61] dB, attack
 2 “sust”:  linf [-24, 24, 97] dB, sustain
 3 “g”:     linf [-18, 9, 55] dB, gain
```

Source Extractor

```
 0 “mdl”:   PSE
 1 “thr”:   linf [-36, 12, 97] dB, threshold
 2 “depth”: linf [0, 20, 41] dB, depth
 3 “fast”:  int [0, 1] fast
 4 “peak”:  int [0, 1] peak
```

PSE/LA Combo

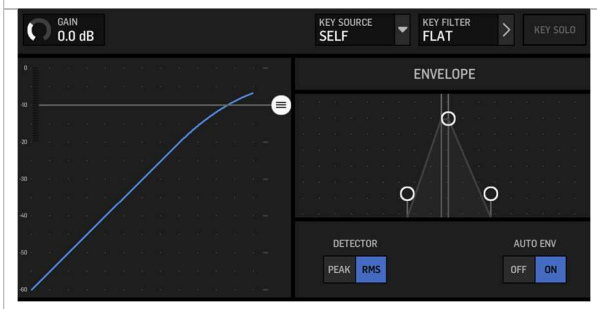
```
 0 “mdl”:   CMB
 1 “thr”:   linf [-36, 12, 97] dB, threshold
 2 “depth”: linf [0, 20, 41] dB, depth
 3 “fast”:  int [0, 1] fast
 4 “peak”:  int [0, 1] peak
 5 “ingain”:linf [0, 100, 101] gain
 6 “peak”:  linf [0, 100, 101] peak
 7 “mode”:  str [comp, lim] mode
```

Auto Rider Dynamics

```
 0 "mdl":   RIDE
 1 "thr":   linf [-54, 18, 73] dB, thr
 2 "tgt":   linf [-48, 0, 97] dB, target
 3 "spd":   int [1…50] speed
 4 "ratio": flt [2.0, 4.0, 8.0,
                 20.0, 100.0] ratio
 5 "hld":   logf [.1, 10, 65] s, hold
 6 "range": linf [1, 15, 29] dB, range
```

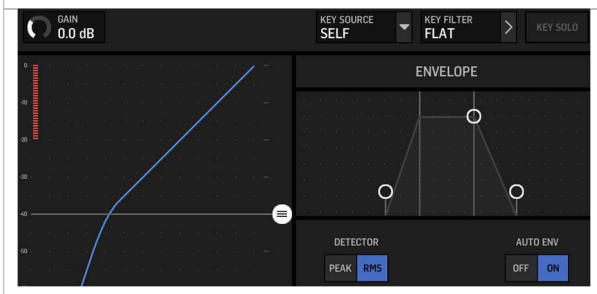
Soul Warmth Preamp

```
 0 "mdl":   WARM
 1 "drv":   linf [10, 100, 91] %, drive
 2 "hrm":   linf [-100, 100, 201] harm
 3 "col":   linf [-1, 1, 41] color
 3 "trim":  linf [-18, 6, 49] dB, trim
 4 "mix":   linf [0, 100, 101] dB, mix
```



WING compressor

```
 0 "mdl":   COMP
 1 "mix":   linf [0, 100, 101] %, mix
 2 "gain":  linf [-6, 12, 37] dB, gain
 3 "thr":   linf [-60, 0, 121] dB, thr
 4 "ratio": flt [1.1, 1.2, 1.3, 1.5, 1.7, 2.0,
                 2.5, 3.0, 3.5, 4.0, 5.0, 6.0,
                 8.0, 10., 20., 50., 100.] ratio
 5 "knee":  int [0…5] knee
 6 "det":   str [PEAK, RMS] detector
 7 "att":   linf [0, 120, 121] ms, attack
 8 "hld":   linf [1, 200, 200] ms, hold
 9 "rel":   logf [4, 4000, 121] ms release
10 "env":   str [LIN, LOG] envelope
11 "auto":  int [0, 1] auto
```



Wing Expander

```
 0 "mdl":   EXP
 1 "mix":   linf [0, 100, 101] %, mix
 2 "gain":  linf [-6, 12, 37] dB, gain
 3 "thr":   linf [-60, 0, 121] dB, thr
 4 "ratio": flt [1.1, 1.2, 1.3, 1.5, 1.7, 2.0,
                 2.5, 3.0, 3.5, 4.0, 5.0, 6.0,
                 8.0, 10., 20., 50., 100.] ratio
 5 "knee":  int [0…5] knee
 6 "det":   str [PEAK, RMS] detector
 7 "att":   linf [0, 120, 121] ms, attack
 8 "hld":   linf [1, 1000, 121] ms, hold
 9 "rel":   logf [4, 4000, 121] ms release
10 "env":   str [LIN, LOG] envelope
11 "auto":  int [0, 1] auto
```

BDX 160 Compressor/Limiter

```
 0 "mdl":   B160
 1 "mix":   linf [0, 100, 101] %, mix
 2 "gain":  linf [-6, 12, 37] dB, gain
 3 "thr":   logf [.01, 5, 65] thr
 4 "ratio": flt [1.1, 1.2, 1.3, 1.5, 1.7, 2.0,
                 2.5, 3.0, 3.5, 4.0, 5.0, 6.0,
                 8.0, 10., 20., 50.] ratio
```

BDX 560 Easy Compressor

```
 0 "mdl":   B560
 1 "mix":   linf [0, 100, 101] %, mix
 2 "gain":  linf [-6, 12, 37] dB, gain
 3 "thr":   linf [-40, 20, 121] dB, thr
 4 "ratio": flt [1.1, 1.2, 1.5, 2.0, 3.0, 4.0,
```

	5.0, 7.0, 10., 50., 999., -5.0, -3.0, -2.0, -1.0] ratio 5 **"auto"**: int [0, 1] auto
	Draw More Compressor 0 **"mdl"**: D241C 1 **"mix"**: linf [0, 100, 101] %, mix 2 **"gain"**: linf [-6, 12, 37] dB, gain 3 **"thr"**: linf [0, -60, 121] dB, thr 4 **"ratio"**: flt [1.1, 1.2, 1.3, 1.5, 1.7, 2.0, 3.0, 3.5, 4.0, 5.0, 6.0, 8.0, 10.0, 20.0, 50.0, 100.0] ratio 5 **"att"**: linf [.5, 100, 65] ms, attack 6 **"rel"**: logf [50, 5000, 130] ms release 7 **"lim"**: linf [-20, 0, 41] dB, lim thr 8 **"lrel"**: logf [50, 5000, 130] ms, lim rel 9 **"auto"**: int [0, 1] auto
	Even Compressor/Limiter 0 **"mdl"**: ECL33 1 **"mix"**: linf [0, 100, 101] %, mix 2 **"gain"**: linf [-6, 12, 37] dB, gain 3 **"lon"**: int [0, 1] lim on 4 **"lthr"**: linf [-12, 0, 25] dB, lim thr 5 **"lrec"**: str [50, 100, 200, 800, A1, A2] lim rec 6 **"lfast"**: int [0, 1] lim fast 7 **"con"**: int [0, 1] comp on 8 **"cthr"**: linf [-35, -5, 61] dB, comp thr 9 **"ratio"**: str [1.5, 2.0. 3.0, 4.0, 6.0] ratio 10 **"crec"**: str [100, 400, 800, 1500 A1, A2] comp rec 11 **"cfast"**: int [0, 1] comp fast
	Soul 9000 Channel Compressor 0 **"mdl"**: 9000C 1 **"mix"**: linf [0, 100, 101] %, mix 2 **"gain"**: linf [-6, 12, 37] dB, gain 3 **"thr"**: linf [-48, 0, 97] dB, thr 4 **"ratio"**: flt [1.3, 1.43, 1.57, 1.8, 2.0, 2.8, 3.3, 4.0, 5.0 , 6.0, 7.0, 9.0, 12.0, 20.0, 50.0, 100.0] ratio 5 **"fast"**: int [0, 1] fast att 6 **"rel"**: logf [100, 4000, 65] ms release 7 **"peak"**: int [0, 1] peak
	Soul G Buss Compressor 0 **"mdl"**: SBUS 1 **"mix"**: linf [0, 100, 101] %, mix 2 **"gain"**: linf [-6, 12, 37] dB, gain 3 **"thr"**: linf [-48, 0, 81] dB, thr 4 **"ratio"**: flt [1.5, 2.0, 3.0, 4.0, 5.0, 10.0] ratio 5 **"att"**: flt [0.1, 0.3, 1.0, 3.0, 10.0, 30.0] ratio 6 **"rel"**: str [0.1, 0.2, 0.4, 0.8, 1.6, AUTO] release

Red Compressor

```
 0 "mdl":   RED3
 1 "mix":   linf [0, 100, 101] %, mix
 2 "gain":  linf [-6, 12, 37] dB, gain
 3 "thr":   linf [-48, 0, 97] dB, thr
 4 "ratio": flt [1.1, 1.2, 1.3, 1.5, 2.0,
                 2.5, 3.0, 3.5, 4.0, 5.0,
                 6.0, 8.0, 10.] ratio
 5 "att":   linf [1, 50, 65] ms, attack
 7 "rel":   logf [100, 4000, 65] ms release
 8 "auto":  int [0, 1] auto
```

76 Limiting Amplifier

```
 0 "mdl":   76LA
 1 "mix":   linf [0, 100, 101] %, mix
 2 "gain":  linf [-6, 12, 37] dB, gain
 3 "in":    linf [-48, 0, 97] dB, input
 4 "out":   linf [-48, 0, 97] dB
 5 "att":   linf [1, 7, 61] attack
 6 "rel":   linf [1, 7, 61] release
 7 "ratio": str [4, 8, 12, 20, ALL] ratio
```

Leveling Amplifier 2A

```
 0 "mdl":   LA
 1 "ingain":linf [0, 100, 101] gain
 2 "peak":  linf [0, 100, 101] peak
 3 "mode":  str [comp, lim] mode
```

Fairkid Model 670

```
 0 "mdl":   F670
 1 "mix":   linf [0, 100, 101] %, mix
 2 "gain":  linf [-6, 12, 37] dB, gain
 3 "in":    linf [-20, 0, 81] dB, input
 4 "thr":   linf [0, 10, 41] thr
 5 "time":  int [1…6] time
 6 "bias":  linf [0, 1, 101] bias
```

Eternal Bliss

```
 0 "mdl":   BLISS
 1 "mix":   linf [0, 100, 101] %, mix
 2 "gain":  linf [-6, 12, 37] dB, gain
 3 "thr":   linf [-50, 0, 101] dB, thr
 4 "ratio": flt [1.2, 1.3, 1.6, 2.0, 3.0,
                 -1.0, -2.0, -3.0,  -4.0] ratio
 5 "att":   linf [.4, 150, 65] ms, attack
 6 "rel":   logf [5, 1200, 65] ms release
 7 "afast": int [0, 1] auto fast
 8 "alog":  int [0, 1] anti log
 9 "glon":  int [0, 1] gr limit on
10 "glim":  linf [-21, 0, 43] gr limit
```

No Stressor

```
 0 "mdl":   NSTR
 1 "mix":   linf [0, 100, 101] %, mix
 2 "gain":  linf [-6, 12, 37] dB, gain
 3 "in":    linf [0, 10, 101] input
 4 "ou":    linf [0, 10, 101] output
 5 "att":   linf [0, 10, 101] attack
 6 "rel":   linf [0, 10, 101] release
 7 "ratio": str [1.5:1, 2:1, 3:1, 4:1, 6:1,
                 10:1, 20:1, NUKE] ratio
```

PIA 2250

*0 “**mdl**”: 2250*
*1 “**mix**”: linf [0, 100, 101] %, mix*
*2 “**gain**”: linf [-6, 12, 37] dB, gain*
*3 “**thr**”: linf [0, 10, 101] threshold*
*4 “**ratio**”: linf [0, 10, 101] output*
*5 “**att**”: str [FAST, MED, SLOW] attack*
*6 “**rel**”: logf [50, 3000, 130] ms, release*
*7 “**knee**”: str [HARD, SOFT] knee*
*8 “**Type**”: str [OLD, NEW] type*

LTA100 Leveler

*0 “**mdl**”: L100*
*1 “**mix**”: linf [0, 100, 101] %, mix*
*2 “**gain**”: linf [-6, 12, 37] dB, gain*
*3 “**ingain**”:linf [0, 10, 101] gain*
*4 “**gr**”: linf [0, 10, 101] gain reduction*
*5 “**att**”: str [FAST, MED, SLOW] attack*
*6 “**rel**”: str [FAST, MED, SLOW] release*

Even 88 Compressor

*0 “**mdl**”: E88C*
*1 “**mix**”: linf [0, 100, 101] %, mix*
*2 “**gain**”: linf [-6, 12, 37] dB, gain*
*3 “**knee**”: str [SOFT, HARD] knee*
*4 “**thr**”: linf [20, -10, 61] dB threshold*
*5 “**thrpull**”: int [0..1], -20dB pull*
*6 “**ratio**”: logf [1, 10, 101] ratio*
*7 “**att**”: str [SLOW, FAST] attack*
*8 “**rel**”: logf [10,3000,201] ms, release*

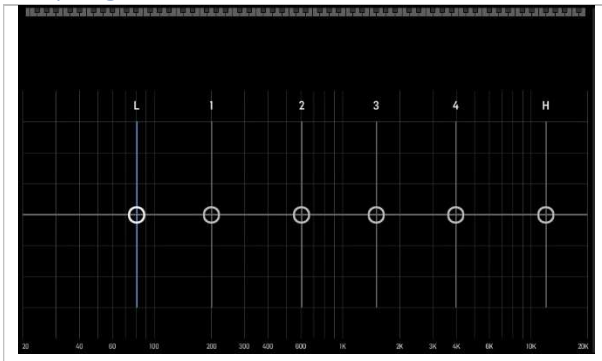
LMT Compressor

*0 “**mdl**”: LMT*
*1 “**mix**”: linf [0, 100, 101] %, mix*
*2 “**gain**”: linf [-6, 12, 37] dB, gain*
*3 “**tspd**”: linf [0, 1, 101] speed*
*4 “**trans**”: linf [-10, 10, 101] transients*
*5 “**tgain**”: linf [-12, 12, 241], dB trans gain*
*6 “**ton**”: int [0..1] shaper on*
*7 “**comp**”: linf [0, 100, 101] comp*
*8 “**cgain**”: linf [-12, 12, 241] comp gain*
*9 “**con**”: int [0..1] comp on*

One Knob Compressor

*0 “**mdl**”: ONEC*
*1 “**mix**”: linf [0, 100, 101] %, mix*
*2 “**gain**”: linf [-6, 12, 37] dB, gain*
*3 “**gr**”: linf [0, 10, 101] gain reduction*
*4 “**dag**”: int [0..1] auto gain*

EQ plugins



Standard EQ

Channel:

```
 0 "mdl":  STD
 1 "lg":   linf [-15, 15, 301] dB, gain l
 2 "lf":   logf [20, 2000, 641] Hz, freq l
 3 "lq":   logf [0.442, 10, 181] q l
 4 "leq":  str [SHV, PEQ] eq l
 5 "1g":   linf [-15, 15, 301] dB, gain 1
 6 "1f":   logf [20, 20000, 961] Hz, freq 1
 7 "1q":   logf [0.442, 10, 181] q 1
 8 "2g":   linf [-15, 15, 301] dB, gain 2
 9 "2f":   logf [20, 20000, 961] Hz, freq 2
10 "2q":   logf [0.442, 10, 181] q 2
11 "3g":   linf [-15, 15, 301] dB, gain 3
12 "3f":   logf [20, 20000, 961] Hz, freq 3
13 "3q":   logf [0.442, 10, 181] q 3
14 "4g":   linf [-15, 15, 301] dB, gain 4
15 "4f":   logf [20, 20000, 961] Hz, freq 4
16 "4q":   logf [0.442, 10, 181] q 4
17 "hg":   linf [-15, 15, 301] dB, gain h
18 "hf":   logf [50, 20000, 833] Hz, freq h
19 "hq":   logf [0.442, 10, 181] q h
20 "heq":  str [SHV, PEQ] eq h
```

Bus, mtx, main:

```
 0 "mdl":  STD
 1 "lg":   linf [-15, 15, 301] dB, gain l
 2 "lf":   logf [20, 2000, 641] Hz, freq l
 3 "lq":   logf [0.442, 10, 181] q l
 4 "leq":  str [SHV, PEQ, CUT] eq l
 5 "1g":   linf [-15, 15, 301] dB, gain 1
 6 "1f":   logf [20, 20000, 961] Hz, freq 1
 7 "1q":   logf [0.442, 10, 181] q 1
 8 "2g":   linf [-15, 15, 301] dB, gain 2
 9 "2f":   logf [20, 20000, 961] Hz, freq 2
10 "2q":   logf [0.442, 10, 181] q 2
11 "3g":   linf [-15, 15, 301] dB, gain 3
12 "3f":   logf [20, 20000, 961] Hz, freq 3
13 "3q":   logf [0.442, 10, 181] q 3
14 "4g":   linf [-15, 15, 301] dB, gain 4
15 "4f":   logf [20, 20000, 961] Hz, freq 4
16 "4q":   logf [0.442, 10, 181] q 4
17 "5g":   linf [-15, 15, 301] dB, gain 5
18 "5f":   logf [20, 20000, 961] Hz, freq 5
19 "5q":   logf [0.442, 10, 181] q 5
20 "6g":   linf [-15, 15, 301] dB, gain 6
21 "6f":   logf [20, 20000, 961] Hz, freq 6
22 "7q":   logf [0.442, 10, 181] q 6
23 "hg":   linf [-15, 15, 301] dB, gain h
24 "hf":   logf [50, 20000, 833] Hz, freq h
25 "hq":   logf [0.442, 10, 181] q h
26 "heq":  str [SHV, PEQ, CUT] eq h
27 "tilt": linf [-6, 6, 49] dB, tilt
```

Soul Analog EQ

```
 0 "mdL":   SOUL
 1 "mix":   linf [0, 125, 126] %, mix
 2 "lf":    linf [0, 10, 101] lo freq
 3 "lg":    linf [-5, 5, 101] lo gain
 4 "lmf":   linf [0, 10, 101] lm freq
 5 "lmf3":  int [0, 1] lm /3
 6 "lmq":   linf [0, 10, 101] lm q
 7 "lmg":   linf [-5, 5, 101] lm gain
 8 "hmf":   linf [0, 10, 101] hm freq
 9 "hmf3":  int [0, 1] hm x3
10 "hmq":   linf [0, 10, 101] hm q
```

	11 **“hmg”:** linf [-5, 5, 101] hm gain 12 **“hf”:** linf [0, 10, 101] hf freq 13 **“hg”:** linf [-5, 5, 101] hf gain
	Even 88-Formant EQ 0 **“mdl”:** E88 1 **“mix”:** linf [0, 125, 126] %, mix 2 **“lf”:** linf [0, 10, 101] lf freq 3 **“lg”:** linf [-5, 5, 101] lf gain 4 **“lq”:** str [LOW, HIGH] lf q 5 **“lt”:** str [BELL, SHELV] lf type 6 **“lmf”:** linf [0, 10, 101] lm freq 7 **“lmg”:** linf [-5, 5, 101] lm gain 8 **“lmq”:** linf [0, 10, 101] lm q 9 **“hmf”:** linf [0, 10, 101] hm freq 10 **“hmg”:** linf [-5, 5, 101] hm gain 11 **“hmq”:** linf [0, 10, 101] hm q 12 **“hf”:** linf [0, 10, 101] hm freq 13 **“hg”:** linf [-5, 5, 101] hf gain 14 **“hq”:** str [LOW, HIG] hf q 15 **“ht”:** str [BELL, SHELV] hf type
	Even 84 EQ 0 **“mdl”:** E84 1 **“mix”:** linf [0, 125, 126] %, mix 2 **“g”:** linf [-20, 20, 81] dB, gain 3 **“lf”:** str [OFF, 35, 60, 110, 220] lf freq 4 **“lg”:** linf [-5, 5, 101] lf gain 5 **“mf”:** str [OFF, 350, 700, 1k6, 3k2, 4k8, 7k2] mid freq 6 **“mg”:** linf [-5, 5, 101] mid gain 7 **“mq”:** str [LOW, HIGH] mid q 8 **“hf”:** str [10k, 12k, 16k, OFF] hf freq 9 **“hg”:** linf [-5, 5, 101] hf gain
	Focusrite ISA 110 EQ 0 **“mdl”:** F110 1 **“mix”:** linf [0, 125, 126] %, mix 2 **“peq”:** int [0, 1] peq on 3 **“lmf”:** linf [0, 10, 101] lm freq 4 **“lmg”:** linf [-5, 5, 101] lm gain 5 **“lmq”:** linf [0, 10, 101] lm q 6 **“lmf3”:** int [0, 1] lm /3 7 **“hmf”:** linf [0, 10, 101] hm freq 8 **“hmg”:** linf [-5, 5, 101] hm gain 9 **“hmq”:** linf [0, 10, 101] hm q 10 **“hmf3”:** int [0, 1] hm x3 11 **“shv”:** inf [0, 1] shv on 12 **“lf”:** str [33, 56, 95, 160, 270, 460] lf freq 13 **“lg”:** linf [-5, 5, 101] lf gain 14 **“hf”:** str [3k3, 4k7, 6k8, 10k, 15k, 18k] hf freq 15 **“hg”:** linf [-5, 5, 101] hf q 16 **“g”:** linf [-18, 18, 73] gain
	Pulsar P1a/M5 EQ 0 **“mdl”:** PULSAR 1 **“mix”:** linf [0, 125, 126] %, mix 2 **“eq1”:** int [0, 1] eq1 on 3 **“1lb”:** linf [0, 10, 101] lf boost 4 **“1latt”:** linf [0, 10, 101] lf att 5 **“1lf”:** str [20, 30, 60, 100] Hz, lf freq 6 **“1hw”:** linf [0, 10, 101] hf wid 7 **“1hb”:** linf [0, 10, 101] hf boost

	8 **“1hf”**: str [3k, 4k, , 5k, 8k, 10k, 12k, 16k] Hz, hf freq 9 **“1hatt”**: linf [0, 10, 101] hf att 10 **“1hattf”**:str [5k, 10k, 20k] hf att 11 **“eq5”**: int [0, 1] eq5 on 12 **“5lb”**: linf [0, 10, 101] lm boost 13 **“5lf”**: str [200, 300, 500, 700, 1k] Hz, lf freq 14 **“5md”**: linf [0, 10, 101] mid dip 15 **“5mf”**: str [200, 300, 500, 700, 1k, 1k5, 2k, 3k, 4k, 5k, 7k] Hz, mid freq 16 **“5hb”**: linf [0, 10, 101] HM boost 17 **“5hf”**: str [1k5, 2k, 3k, 4k, 5k] Hz, hf freq
	Mach EQ4 0 **“mdl”**: MACH4 1 **“mix”**: linf [0, 125, 126] %, mix 2 **“sub”**: linf [-5, 5, 101] sub 3 **“40”**: linf [-5, 5, 101] 40 4 **“160”**: linf [-5, 5, 101] 160 5 **“650”**: linf [-5, 5, 101] 650 6 **“2k5”**: linf [-5, 5, 101] 2k5 7 **“air”**: linf [0, 10, 101] air 8 **“airm”**: str [OFF, 2k5, 5k, 10k, 20k, 40k] air mode 9 **“again”**: int [0, 1] auto

Appendix: WING Effects Description[108]

In May 2025, Dana Tucker[109] used AI, contents from the `Behringer.world`[110] forum and other sources to generate a description of some of WING effects. The result is reproduced here with permission from the author.

This appendix provides a detailed description of some of the WING effects and effects parameters, and when possible, the application and model used as the origin of the effect; Indeed, many of these Effects (FX) are emulated after very high-end outboard models that have been proven over time. We provide here as much information as possible to help explain what each FX is used for, as well as who used them. The goal is to help understand what they do.

This guide is nothing more than a starting point for anyone who may be reading/using it. Like anything else in audio, you really need to spend some time playing around with these effects so you can see just how much they can enhance your style of music or the way you use your mixer.

In audio mixing, effects are used to shape, enhance, and transform audio signals. There are numerous types of effects, and they can be broadly categorized based on their function. Here are some of the most common types of audio effects:

Time-Based Effects:

- Reverb: Adds spatial depth and ambiance by simulating reflections in a space.
- Delay: Creates echoes by repeating the sound after a set time.
- Chorus: Simulate the subtle pitch and timing differences that occur when multiple musicians or vocalists play the same note.
- Flanger: A swirling, "jet-like" sound by mixing two signals together, with one signal slightly delayed and modulated in phase.
- Phaser: Modulation effect that alters the phase of an audio signal.
- Echo: A specific type of delay with feedback.

Modulation Effects:

- Chorus
- Flanger
- Phaser
- Vibrato: Pitch modulation effect at constant speed.
- Tremolo: Amplitude (volume) modulation.

Dynamic Effects:

- Compression: Reduces the dynamic range by attenuating loud sounds.
- Limiter: Prevents signals from exceeding a certain threshold.
- Expansion: Opposite of compression, increases dynamic range.
- Gating: Cuts off sound below a certain threshold.

Filter Effects:

[108] This chapter is likely to disappear in the future, as Behringer is creating a proper User Manual for WING that contains an FX description section. See: https://www.behringer.com/product.html?modelCode=0603-AEN, under Documentation
[109] www.danatucker.com – Thanks for the authorization to copy this material
[110] https://behringer.world

- Equalization (EQ): Adjusts the balance of frequency components.
- High-pass filter: Removes low frequencies.
- Low-pass filter: Removes high frequencies.
- Band-pass filter: Allows only a specific frequency band.
- Notch filter: Removes a narrow frequency range.

Distortion and Saturation Effects:

- Overdrive: Adds harmonic distortion for a warm or aggressive sound.
- Distortion: Creates more intense tonal changes, often used in guitars.
- Fuzz: Extreme distortion for a fuzzy sound.

Specialty Effects:

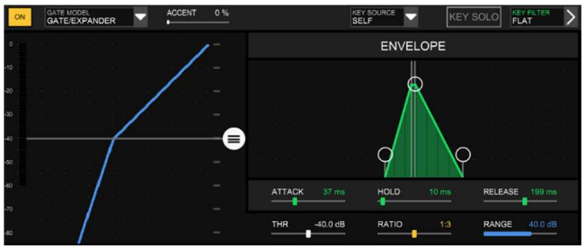
- Auto-tune / Pitch correction: Corrects or alters pitch.
- BitCrusher: Reduces bit depth, creating a lo-fi sound.
- Ring Modulation: Combines two signals to produce metallic or bell-like sounds.
- Synthesizer effects: Creating sounds via synthesis.

Creative and Experimental Effects:

- Granular synthesis effects.
- Reverse effects: Playing sounds backward.
- Spectral effects: Manipulating the frequency spectrum directly.

Gate Section

Wing Gate/Expander



The Behringer Wing Gate/Expander is known for its versatile and high-quality dynamics processing capabilities integrated into the Behringer Wing digital mixing console. Specifically, it is recognized for:

Comprehensive Dynamics Control: The gate/expander allows precise control over audio signals by reducing or eliminating unwanted noise, feedback, or background sounds when the input falls below a certain threshold.

Intuitive User Interface: The feature is accessible via the console's touchscreen and dedicated controls, making it user-friendly for both novice and experienced engineers.

Flexibility and Precision: With adjustable parameters such as threshold, ratio, attack, release, and hold, users can finely tune the gating or expansion to suit various applications—from live sound reinforcement to studio recording.

Integrated with the Digital Mixer: Because it's built into the Wing's comprehensive digital ecosystem, it can be linked with other effects and processing modules for streamlined workflow.

High-Quality Sound Performance: Behringer is known for providing professional-grade audio processing at a more accessible price point, and the Gate/Expander on the Wing maintains clarity and transparency in dynamics control.

Overall, the Behringer Wing Gate/Expander is known for offering powerful, flexible, and easy-to-use gating and expansion functions within a professional digital mixing environment.
Attack is the time taken for the gate to open after an over threshold signal. The shape of the attack is fixed and has been carefully tailored to produce a transparent gating action.
Hold minimizes chattering in conjunction with the internal hysteresis. Once the signal has been detected as having fallen below threshold, this control defines a waiting period before the gate starts to close. This is particularly useful for low frequency material and instruments with oscillating or unpredictable decay envelopes.
Release is the time taken for the gate to close after the program material falls back below threshold. As with attack, the shape is crucial to the sound and has been tailored to produce the most transparent gating action possible.
Range controls the amount of gain reduction that is applied to signals below threshold. The gain reduction can be infinite, but things often sound more natural when it is backed off to only 10dB or 15dB. With this type of setting, the background noise and spill remain at a reduced level, but become less noticeable because they do not noticeably switch in and out with the gating action. The maximum range is given when the control is fully anti-clockwise, that is, set to infinity.
Threshold is the gate operating point. Signals that go over threshold will open the gate, while signals that go below threshold will close the gate. In both cases, gate opening/closing occurs over a period of time, which is dependent on the envelope (attack and release) control settings.
Accent. The inspiration of the gate-expander accent control comes from the Klark Teknik DN530 Quad Gate. It accents (boosts) the transient signal as the gate opens for a brief 50 milliseconds, which is useful for percussion type signals, such as snare, kick, etc. Normally, during an opening transition the gain changes from -n dB (n is set up by the range control) to 0dB. When the accent control is turned up the transition goes from -n dB to a positive +n dB gain (the amount of positive gain is set by the accent control up to a maximum of +12dB). This accented level only lasts for a short period of time (50ms), after which the gain returns to 0dB. The effect this produces is similar to the thickening action compressors impart on drums when their attack time is set very slow.

Soul 9000 Gate. Emulates the SSL 9000 Channel Gate

The SSL 9000 Channel Gate is a renowned gate plugin that emulates the noise gate and expansion features of the Solid State Logic (SSL) 9000 series console's channel strip. It is known for its high-quality, musical gating capabilities, allowing engineers to control unwanted noise, tighten up recordings, and add punch to individual tracks.

The SSL 9000 Channel Gate is appreciated for its transparent sound, intuitive interface, and its ability to preserve the natural character of the audio while effectively gating unwanted signals, making it a popular choice in mixing and mastering workflows.
The Gate can act as an infinite:1 Gate or as a 2:1 Expander when the 2:1 button is engaged. A yellow LED indicates that Expand mode has been selected.

Range determines the depth of Gating or Expansion. When set to 0, this section is inactive. When turned fully clockwise, a Range of 40dB can be obtained.
Threshold – Variable hysteresis is incorporated in the Threshold circuitry. For any given 'open' setting, the Expander/Gate will have a lower 'close' threshold. The hysteresis value is increased as the threshold is lowered. This is very useful in music recording as it allows instruments to decay below the open threshold before Gating or Expansion takes place.
Release – This determines the time constant (speed), variable from 0.1 - 4 seconds, at which the Gate/Expander reduces the signal level once it has passed below the threshold.

The Fast attack button provides a fast attack time (100µs per 40db) indicated with a yellow active LED. When off, a controlled linear attack time of 1.5ms per 40dB is
selected. The attack time is the time taken for the Expander/Gate to 'recover' once the signal is above the threshold. When gating signals with a steep rising edge, such as drums, a slow attack may effectively mask the initial THWACK, so you should be aware of this when selecting the appropriate attack time.
Hold determines the time after the signal decays below the threshold before the gate closes. Variable from 0 to 4 seconds.

Even 88 Gate. Emulates the Neve 88RS Gate.

The Neve 88RS Gate is renowned for its high-quality, musical gating capabilities, integral to the classic Neve 88RS console's reputation. It is known for:
Transparent and Musical Gating: The gate provides smooth, musical attenuation that preserves the natural character of the sound, making it ideal for controlling bleed and unwanted noise without sounding obvious or choppy.
High-Quality Sound: Built with the renowned Neve transformer design and high-grade components, it offers a warm, sonically pleasing attenuation that complements the Neve console's overall sonic signature.
Versatility: The gate is versatile, suitable for a wide range of applications, including drums, vocals, and instruments, allowing for precise control over audio dynamics.
Integral to the 88RS Console: As part of the Neve 88RS console's channel strip, it contributes to the console's reputation for delivering professional-grade, warm, and musical sound processing.
Overall, the Neve 88RS Gate is celebrated for its blend of transparency, musicality, and classic Neve character, making it a sought-after piece of equipment both in its original form and as a standalone component in modern studio setups.
In the context of audio gates, hysteresis refers to the difference between the threshold
level at which a gate opens and the threshold level at which it closes. This helps prevent "chattering" or unwanted opening and closing of the gate when the signal fluctuates around the threshold.

Hysteresis can help preserve the natural sound of vocals or instruments, especially when they naturally fluctuate in volume. It's particularly useful for signals that have subtle changes in volume, preventing the gate from opening and closing repeatedly and can be fine-tuned for very subtle differences.

Draw More 241. Emulates the Drawmer DL241 Expander/Gate Section.

The Drawmer DL241 Expander/Gate Section is renowned for its high-quality, versatile dynamics processing capabilities. It is particularly recognized for:
Transparent Dynamic Control: The DL241 provides clean and transparent gating and expansion, allowing engineers to control noise and leakage without introducing unwanted artifacts.
Dual-Channel Operation: Designed for stereo or dual-mono applications, it offers precise, independent control over each channel, making it suitable for complex mixing and mastering tasks.
High-Quality Components and Design: Built with professional-grade circuitry, the DL241 ensures reliable performance and minimal signal coloration.
Flexible Threshold and Ratio Controls: It allows detailed adjustment of gating and expansion parameters, enabling both subtle noise reduction and aggressive gating when needed.
User-Friendly Interface: The layout and controls are designed for intuitive operation, facilitating quick setup and adjustments during critical recording or mixing sessions.

Overall, the Drawmer DL241 is valued for its clarity, flexibility, and professional-grade performance in controlling dynamics within a studio or live sound environment.

BDX 902 De-Esser. Emulates the DBX 902.

The BDX 902 De-Esser is known for being a high-quality audio processing device designed to reduce sibilance (the harsh "s" and "sh" sounds) in vocal recordings. It is particularly valued in professional audio production for its precise and transparent
de-essing capabilities, helping to produce clearer, more natural-sounding vocals without sacrificing detail.

The BDX 902 often features advanced filtering and control options, making it a preferred choice for engineers seeking effective de-essing in both studio and broadcast environments.

76 Limiter Amp. Emulates the UREI/Universal Audio 1176 FET Compressor.

The UREI/Universal Audio 1176 FET Compressor is renowned for its distinctive sound and versatility in audio processing. It is widely regarded as one of the most iconic and sought-after compressors in professional recording and mixing. Key features and qualities it is known for include:
Fast Attack and Release Times: The 1176 is famous for its incredibly quick attack and release times, allowing it to catch transient peaks and shape dynamic responses sharply.

Unique "All-Button" Mode: Engaging all ratio buttons simultaneously creates a distinctive "British" style compression—characterized by a heavily saturated, harmonically rich sound.
FET Technology: Its use of Field Effect Transistors (FETs) provides a distinctive tonal character, often described as aggressive, punchy, and musical.
Musical and Transparent Compression: Depending on settings, it can deliver transparent leveling or a more colored, colored compression that adds character and mojo to vocals, drums, bass, and other sources.
Versatility: The 1176 excels across a wide range of audio sources—vocals, drums, bass, guitars, and mix bus compression—making it a studio staple.
Historical Significance: Originally designed in the 1960s by Bill Putnam's UREI company, the 1176 became a standard piece of equipment in studios worldwide and has influenced countless compressor designs.

Overall, the UREI/Universal Audio 1176 FET Compressor is celebrated for its ability to add punch, presence, and musical character to recordings, making it a legendary tool in both tracking and mixing contexts.
Great on drums, vocals, bass, parallel compression. The Threshold is fixed, Input drives level up into the threshold, output gain compensates for compression and possible input increase. It has a very fast attack and release. Be very careful as the timing knobs are reversed from normal with 1 being the slowest and 7 being the fastest.

LA Leveler. Emulates the Teletronix LA-2A.

The Teletronix LA-2A is renowned for its smooth, musical, and transparent optical compression characteristics. It is widely regarded as a classic leveling amplifier, prized for its ability to gently control dynamics without introducing harsh artifacts.

The LA-2A is especially favored for vocals, bass, and other instruments where natural, transparent compression is desired. Its unique design uses an electro-optical attenuator and a tube-based amplifier, resulting in a warm, musical sound that has made it a staple in professional recording and broadcast studios worldwide.
It is well known for its warm optical compressor and works great on vocals, bass, horns and strings. It has fixed slow attack, slow release & ratio. The compression is 3:1 and the limit is infinite:1. You turn up Peak Reduction to increase the amount of compression, essentially a reverse threshold. The gain is post-compression makeup gain and the VU meter trails actual compression.

Source Extractor. Emulates the Rupert Neve Primary Source Enhancer PSE-545.

The Rupert Neve Primary Source Enhancer (PSE-545) is renowned for its ability to improve the clarity, presence, and detail of vocal and instrumental recordings. It is a specialized audio processor designed to enhance the primary source signals—such as vocals or solo instruments—by emphasizing their character and reducing background noise or masking elements. Key features and qualities include:

Enhanced Clarity and Presence: The PSE-545 accentuates the fundamental frequencies and harmonics of the primary source, making it sound more prominent and lifelike in a mix.
Selective Enhancement: It intelligently emphasizes the desired source without overly affecting the surrounding audio, preserving naturalness.
Analog Circuit Design: Built with Rupert Neve's signature high-quality analog circuitry, it offers a warm, musical character to the processed signals.
Application Flexibility: Commonly used in recording and mixing environments to improve vocal intelligibility, instrument definition, and overall mix clarity.
Controls Include:

Active Light: Indicates when the 545 is actively affecting signal. This is the key to using this processor.
Threshold Knob: Sets the dBu level at which the Primary Source Extractor engages.

Depth Knob: Controls the maximum amount of effect from the 545 – or in other words, how much attenuation is applied after the input signal falls below the set THRESHOLD.

By rotating the DEPTH control clockwise from 0dB towards -20dB, the 545 will attenuate the input signal more dramatically, letting the user find the perfect balance between audibility of the Primary Source Enhancement effect and feedback reduction.
Fast Control: Is an illuminated push-button that selects between two available time constants, tuning the 545's response by determining how quickly the attenuation occurs in the quiet sections between words or phrases. Without this button pressed, the 545 achieves a slower attack and release that is useful as a starting point for most sources. Pressing the FAST button engages the faster attack and release time constant, Fast is useful on more dynamic sources or passages where faster transient detection is necessary.
Peak: Selects between RMS and Peak detection modes. When illuminated, the 545 is in PEAK mode – useful for detecting faster transient peaks and for creative dynamic envelope shaping. When not illuminated, the 545 is in RMS mode, which utilizes a slower, more averaged response characteristic in the sidechain.
Overall, the PSE-545 is valued by engineers and producers for its ability to subtly and effectively enhance the perceived quality of key audio sources, resulting in clearer and more engaging recordings.

Wave Designer. Emulates the SPL Transient Designer.

The SPL Transient Designer is renowned for its ability to manipulate the transient and sustain characteristics of audio signals. Specifically, it allows producers and engineers to shape the attack (initial transient) and decay (sustain) of sounds independently.

This makes it a powerful tool for enhancing punch and clarity in drums and percussion, or for smoothing out or emphasizing transients in various instruments. Its unique approach provides a more natural and musical way to control dynamics compared to traditional compressors, making it a popular choice in mixing and sound design.

A full manual is available at https://spl.audio/wp-content/uploads/transient_designer_2_9946_manual.pdf

Auto Rider. Emulates the Waves Vocal Rider

Waves Vocal Rider is known for its ability to automatically and transparently adjust vocal levels within a mix. It functions as an intelligent leveling plugin that dynamically rides the fader in real-time, ensuring consistent vocal volume without the need for manual automation.

This helps producers and engineers achieve a balanced, professional-sounding vocal track more efficiently by saving time and maintaining natural dynamics.

Soul Warmth Pre. Emulates the SSL Console Emulated Preamp.

The SSL Console Emulated Preamp is renowned for its ability to replicate the distinctive sound characteristics of classic SSL analog consoles, particularly their preamp and channel strip qualities. It is known for:

Imitatively Reproducing SSL Sound: Capturing the punch, clarity, and musicality associated with SSL consoles, often favored in professional recording and mixing environments.
Adding Character and Warmth: Providing a sonic coloration that enhances vocals, drums, and other instruments with a lively, punchy, and polished tone.
Ease of Use: Typically offering intuitive controls that emulate the familiar SSL console interface, making it accessible for both novice and experienced engineers.
Versatility: Suitable for a variety of sources, from vocals to drums, giving producers and engineers a quick way to add SSL-style presence and depth to their tracks.
Overall, the SSL Console Emulated Preamp is valued for delivering the signature SSL console sound in a flexible, plugin or hardware form, making it a popular choice for mixing and mastering professionals seeking that classic SSL flavor.

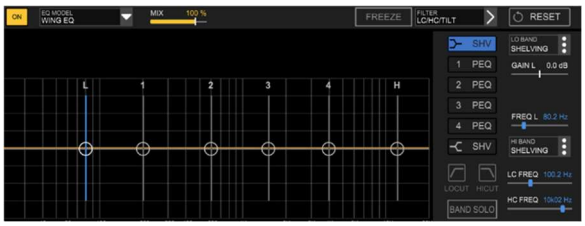
Wing Gate Dynamic EQ

The Wing Gate Dynamic EQ is known for its innovative approach to equalization by combining the traditional functionalities of a parametric EQ with dynamic processing capabilities. It is designed to provide precise, transparent, and musical dynamic EQ adjustments, allowing users to target specific frequency ranges that need dynamic control rather than static adjustments.
Dynamic Filtering: Unlike standard EQs, the Wing Gate Dynamic EQ can respond dynamically to audio signals, reducing or enhancing frequencies only when certain thresholds are exceeded.
Transparency and Musicality: It is praised for maintaining audio clarity while shaping tone, making it suitable for mixing and mastering tasks.
Versatility: Its ability to act as a dynamic EQ, compressor, or de-esser within a single plugin makes it popular among engineers for complex sound shaping.

Ease of Use: Despite its advanced features, it is known for an intuitive interface that allows precise control over dynamic and static EQ settings.
Innovative Design: Wing Gate introduces a unique "gate" mechanism that helps in controlling the dynamic range more effectively, especially useful for controlling resonances, sibilance, or problematic frequencies.
In summary, the Wing Gate Dynamic EQ is known for its flexibility, transparency, and innovative dynamic processing capabilities, making it a powerful tool in professional audio production.

Equalizer Section

Wing EQ



The Behringer Wing EQ is renowned for its high-quality, flexible, and musical equalization capabilities integrated into the Wing digital mixing console. Specifically, it is known for:
Powerful Digital EQs: The Wing features high-resolution, 24-bit/96 kHz digital EQs that provide precise control over sound shaping, allowing for detailed adjustments to individual channels and buses.
Musical Sound Quality: The EQs are designed to be musical and transparent, enabling engineers to enhance clarity, warmth, and presence without introducing harshness or artifacts.
Flexible Filter Types: It offers a variety of filter types, including parametric, shelving, and high/low-pass filters, giving users versatile tools for shaping sound. It also has a fantastic “Tilt” feature.
Intuitive Control Interface: The EQ controls are accessible via the touchscreen interface, physical knobs, and dedicated controls, making real-time adjustments straightforward during live performances.
Integrated Processing: Being part of a digital console, the EQ works seamlessly with other processing features like dynamics and effects, facilitating comprehensive sound optimization.
Overall, the Behringer Wing EQ is known for providing professional-grade, versatile, and user-friendly equalization tools within a compact digital mixing environment, suitable for live sound, touring, and studio applications.

Soul Analog. Emulates the SSL Channel EQ.

The SSL Channel EQ is renowned for its distinctive sound and musical character, primarily because it is modeled after the equalizer found in Solid State Logic (SSL) consoles, particularly the SSL 4000 Series. It is known for:
Musical Tone Shaping: The SSL Channel EQ offers a musical and punchy sound, making it ideal for adding clarity, presence, and warmth to individual tracks or entire mixes.
Flexible Filtering: It provides a combination of high-pass and low-pass filters along with parametric EQ bands, allowing precise control over tonal balance.
Characterful Sound: The EQ is celebrated for its slightly aggressive, yet musical, midrange boost and cut capabilities, which help individual sounds sit well in a mix.
Versatility: It can be used on a wide range of sources—vocals, drums, guitars, and mix buses—thanks to its versatile frequency bands and intuitive interface.
Iconic Status: As a staple in mixing and mastering, the SSL Channel EQ is often emulated in software plugins, and its sound signature is highly regarded in the audio production community.
In summary, the SSL Channel EQ is known for its musical, punchy sound and its ability to enhance clarity and presence in a mix, making it a favorite among engineers for both corrective and creative equalization tasks and is often used on vocals, bass, and acoustic guitars.

Even 88-Formant. Emulates the Neve 88 EQ.

The Neve 88 EQ is renowned for its exceptional sound quality and musical character, making it a favored choice among audio engineers and producers. Specifically, it is known for:

High-Quality Equalization: The Neve 88 EQ offers smooth, musical EQ curves that enhance the warmth and clarity of recordings. Its design allows for precise tonal shaping without introducing harshness.

Classic Neve Sound: It embodies the legendary Neve console sound, characterized by a rich, punchy, and harmonically pleasing tone that is especially prized in recording and mixing.

Versatility: The EQ is versatile, suitable for a wide range of applications—from vocals and guitars to drums and mix buses—adding depth and presence while maintaining musicality.

Analog Warmth: As an analog EQ, it imparts a desirable warmth and character that many digital EQs struggle to replicate authentically.

Overall, the Neve 88 EQ is celebrated for its ability to subtly or dramatically shape sound with a musical, vintage-inspired quality that enhances the emotional impact of recordings.

Product page for the Neve 88 EQ: https://www.ams-neve.com/outboard/88-series-range/8803-2

Even 84. Emulates the AMS Neve 1084 EQ.

The AMS Neve 1084 EQ is renowned for its classic, musical character and high-quality sound, making it a favorite among audio engineers and producers. It is a vintage-style EQ that emulates the legendary Neve 1084 console equalizers, known for their warm, musical tonality and smooth, musical curves. The 1084 EQ is particularly prized for its:

Rich Midrange Shaping: It offers a distinctive, musical midrange boost and cut, ideal for adding warmth and presence to vocals, guitars, and other instruments.

Musical Sound Quality: Its design emphasizes musicality, making it excellent for enhancing recordings without sounding harsh or clinical.

Versatility: Suitable for both tracking and mixing, it can be used on a variety of sources to add character and clarity.

Authentic Vintage Tone: The AMS Neve 1084 EQ captures the essence of the classic Neve console sound, sought after in professional recording and mixing environments.

Overall, the AMS Neve 1084 EQ is celebrated for its ability to impart a warm, musical, and vintage-inspired character to audio, making it a staple in high-end studios.

Nerve Website. https://www.ams-neve.com/outboard/classic-range/1084-2

Fortissimo 110. Emulates the Focusrite ISA 110 EQ.

The Focusrite ISA 110 EQ is renowned for its classic, warm, and musical sound character, making it a popular choice among audio engineers and producers. It is part of the ISA series, inspired by the legendary Focusrite ISA 110 and 130 modules from the
original British console designs. Some of the key features include:

Distinctive Tone: Its EQ section provides a musical and musical-sounding equalization, often described as smooth and musical, helping to enhance recordings without harshness.
Versatility: The ISA 110 offers both high- and low-frequency EQ bands, allowing for precise tonal shaping of vocals, instruments, and mixes.
Vintage Character: The design and circuitry emulate classic British console modules, giving recordings an authentic vintage flavor.
Transparency and Musicality: While it can add character, it also maintains clarity, making it suitable for a variety of sources.
Build Quality: As part of Focusrite's heritage, it is built with high-quality components, ensuring durability and consistent performance.
Overall, the Focusrite ISA 110 EQ is appreciated for its ability to add musicality and character to recordings, making it a sought-after tool for mixing and mastering in both studio and live settings.
Sound on Sound Review. https://www.soundonsound.com/reviews/focusrite-isa-110

Pulsar. Emulates the Pultec EQP-1A combined with MEQ-5.

The Pultec EQP-1A combined with the MEQ-5 is renowned for its unique and highly musical analog equalization characteristics. Some key features are:
Rich, Musical Tone: They impart a warm, smooth, and natural sound that's difficult to replicate with digital EQs.
Smooth, Musical Curves: The Pultec EQP-1A's design allows for gentle, resonant boosts and cuts, creating a "natural" enhancement of frequencies.
Unique Interaction of Frequencies: The Pultec's tube circuitry and passive design enable it to produce resonant peaks and dips that blend seamlessly, often described as "musical" or "musical peak" shaping.
Vintage Character: Both units are iconic in the studio world, associated with classic recordings, especially in mixing for vocals, bass, and drums.
Parallel EQ Curves: The Pultec's ability to boost and attenuate the same frequency simultaneously creates a distinctive "pumping" effect that adds depth and dimension.
Versatility: The combination is used for broad tonal shaping, enhancing warmth, clarity, and presence.
In summary, this combo is celebrated for its ability to add a vintage, musical quality to audio recordings, making it a favorite among mixing engineers seeking that classic analog sound.

Mach EQ4. Emulates the Mäag EQ4.

The Mäag EQ4 is renowned for its unique approach to equalization, primarily its use of a dynamic, "air" boost technique. It is a passive equalizer that employs a special circuitry design to enhance the high-frequency content, often described as adding "air" or "shine" to recordings.

The EQ4 is particularly valued in professional audio for its musical and transparent sound, making it popular for mastering and mixing applications where subtlety and musicality are desired. Its reputation stems from its ability to subtly enhance high-frequency details without introducing harshness, making it a favorite among engineers seeking a natural and refined sound.
The Mäag EQ4 is a well-regarded analog equalizer also known for its musical and musical-sounding tone, often favored in mixing and mastering contexts. Here are some of its key features:
Four-Band Equalizer: The EQ4 offers four bands of equalization—typically low, low-mid, high-mid, and high—allowing precise tonal shaping.
Class-A Circuit Design: It uses Class-A circuitry, which is renowned for its warm, transparent, and musical sound quality.
Variable Frequency & Bandwidth: Each band has adjustable frequency points and bandwidth (Q), giving users flexible control over the tonal adjustments.
Analog Signal Path: The EQ maintains an entirely analog signal path, preserving natural warmth and character.

High-Quality Components: Built with high-grade components to ensure durability and optimal sound quality.
Wide Operating Range: The EQ can handle a broad spectrum of signals, making it suitable for various applications from mixing to mastering.
User-Friendly Interface: Features intuitive controls for easy operation, often with classic VU meters for visual feedback.
Versatility: Suitable for a wide range of audio sources, including vocals, instruments, and full mixes.
Website: https://www.plugin-alliance.com/en/products/maag_eq4.html

PIA 560 GEQ. Emulates: API 560 EQ.

THIS EQ IS ONLY AVAILABLE ON THE BUS, MATRIX AND MAIN CHANNEL STRIPS!

The API 560 EQ, also known as the API 560 Equalizer, is a well-known piece of audio processing equipment recognized for its high-quality, analog equalization capabilities. It is renowned for its:

Vintage Sound Quality: The API 560 EQ imparts a distinctive tonal character often associated with classic analog recordings, characterized by musicality and warmth.
Customizable Bands: It typically features multiple bands of equalization, allowing precise shaping of audio signals across various frequencies.
Robust Construction: Built with durable components, the API 560 is valued for its reliability and longevity in professional studio environments.
Versatility: It is widely used in mixing and mastering to enhance clarity, control tone, and shape the sonic image of individual tracks or entire mixes.

Overall, the API 560 EQ is known for its musical, transparent, and versatile equalization, making it a favorite among audio engineers seeking classic analog character.

Compressor Section

Wing Compressor.

The Behringer Wing Compressor is known for being a versatile and affordable audio processing tool integrated into the Behringer Wing digital mixing console. It is recognized for its high-quality dynamic range control capabilities, allowing users to effectively manage and shape audio signals with precise compression parameters.

The compressor features transparent sound quality, user-friendly controls, and seamless integration within the digital mixer, making it popular among live sound engineers and audio professionals seeking reliable compression without a high cost.

Additionally, its inclusion within the Wing console offers streamlined workflow and advanced processing features suitable for complex live sound and recording applications.
The Behringer Wing Audio Compressor offers a range of features designed to enhance your sound quality and provide precise control over dynamics. Key features include:
High-Quality Compression: Provides transparent and musical compression to control dynamic range and ensure consistent audio levels.
Adjustable Parameters: Control over threshold, ratio, attack, release, and makeup gain to tailor compression to your needs.
Sidechain Functionality: Allows external signal control for more complex compression setups.
Built-In Limiters: Protect your audio signal from clipping and distortion with integrated limiting.

Stereo and Mono Compatibility: Supports both stereo and mono processing for versatile application.
Visual Metering: LED meters provide real-time feedback on gain reduction and output levels.
User-Friendly Interface: Intuitive controls facilitate quick setup and adjustments. XOver (Crossover) Mode Features:
Frequency Division: Splits audio signals into separate frequency bands (e.g., low/mid/high) for targeted processing.
Adjustable Crossover Frequencies: Users can set the crossover points to suit specific audio applications.

BDX 160. Emulates the DBX 160.

The DBX 160 is renowned for its distinctive sound as a classic analog compressor/limiter, widely used in professional audio and music production. It is especially famous for its "warm" and "musical" compression characteristics, which help to add punch and sustain to audio signals.

The DBX 160 is particularly popular for processing drums, bass, and vocals, providing a punchy, aggressive sound that has become a signature in many recordings and live mixes. Its straightforward design and reliable performance have made it a staple in both studio and live sound environments.
Features:

Variable Compression Ratio: Allows precise control over the amount of compression applied to the audio signal, from gentle to heavy compression.
Threshold Control: Sets the level at which compression begins, enabling users to target specific signal peaks.
Attack and Release Times: Adjustable parameters to shape how quickly the compressor responds to signal changes, offering flexibility for different audio sources.
Make-up Gain: Compensates for level reduction caused by compression, ensuring consistent output volume.
Metering: Visual indicators for input level, output level, and gain reduction, helping users monitor the compression process.

Side-Chain Input: For external key signals, enabling techniques like ducking or frequency-specific compression.
Bypass Switch: Allows quick comparison between processed and unprocessed signals.

BDX 560 Easy. Emulates: DBX 560 VCA Overeasy Compressor.

The DBX 560 VCA Overeasy Compressor is renowned for its transparent and musical compression, particularly its unique "Overeasy" compression style. This design allows for smooth, natural-sounding dynamics control that preserves the original character of the audio signal.

It is especially favored in professional audio settings for vocals, drums, and mix bus compression, thanks to its ability to subtly tame peaks while maintaining clarity and punch. The DBX 560's VCA technology combined with the Overeasy circuitry makes it a versatile and reliable tool for achieving polished, dynamic mixes.

The DBX 560 VCA Overeasy Compressor is a professional-grade audio compressor designed to provide transparent and musical dynamic control. Here are some of its key features:
VCA Overeasy Compression: Utilizes VCA (Voltage Controlled Amplifier) technology with Overeasy knee characteristics, allowing for smooth and musical compression transitions.
High-Resolution Metering: Features precise metering for gain reduction and output levels, facilitating accurate adjustments.
Comprehensive Controls Include: Threshold, Ratio, Attack, Release, Make-up Gain & Sidechain Filter (for de-essing or controlling low-frequency compression).

Draw More D241. Emulates the Drawmer DL241.

The Drawmer DL241 is known for being a high-quality, dual-channel, multiband dynamic equalizer and compressor. It is particularly recognized for its precise control over frequency-specific compression and limiting, making it a popular choice in professional audio and mastering environments.

The DL241 offers independent operation of two channels, each with multiple frequency bands, allowing engineers to target specific problem areas in a mix or individual tracks with detailed dynamic processing.

Its reputation stems from its transparent sound quality, flexible controls, and robust build, making it a versatile tool for shaping and controlling audio signals with high precision. Here are some of its key features:
Variable Ratio: Compression ratios adjustable from 1:1 up to 20:1, allowing subtle to aggressive compression.
Threshold Control: Precise threshold adjustment for each channel to set the level at which compression begins.
Attack and Release Times: Fully variable attack (1 ms to 100 ms) and release (50 ms to 2 s) controls for tailoring response.
Limiter Mode: Switchable to act as a peak limiter for transient control.

Side-Chain Filtering: High-pass filter in the side-chain path to prevent low-frequency signals from triggering compression unnecessarily.
Metering: Dual LED meters for each channel to visually monitor gain reduction. Bypass Switch: For A & B comparison.

Red3 Compressor. Emulates: Focusrite Red 3 Compressor.

The Focusrite Red 3 Compressor is renowned for its high-quality, versatile compression capabilities, making it a popular choice among professional audio engineers and producers. It is part of the Focusrite Red series, which is known for its premium analog hardware designed to deliver transparent, musical compression with a rich, detailed sound. Some of its key features are:
High-Fidelity Sound: The Red 3 Compressor offers a transparent and musical compression, preserving the tonal integrity of the source while controlling dynamics effectively.
Versatility: It can be used across a wide range of applications—voice, vocals, instruments, and stereo bus—thanks to its flexible controls and high-quality circuitry.
Precision Control: Features include adjustable attack, release, ratio, and threshold controls, allowing meticulous tailoring of compression to suit various sources.
In summary, the Focusrite Red 3 Compressor is known for delivering transparent, musical compression with a warm analog character, making it a sought-after tool for achieving polished, professional mixes and it is a clean VCA compressor used for mix buses & vocals and it adds a bit of high-mids back into the signal path.

Soul 9000. Emulates the SSL 9000 Channel Compressor.

The SSL 9000 Channel Compressor is renowned for its role in the SSL 9000 Series console, particularly as a high-quality, integrated dynamics processing module. Some of its key features are:
Transparent Compression: The SSL 9000 Channel Compressor provides smooth and transparent compression, making it suitable for a wide range of sources without adding unwanted coloration.
Versatile Dynamics Control: It offers flexible control settings, allowing engineers to precisely shape the dynamics of individual channels, whether vocals, drums, or other instruments.
Integrated Design: As part of the SSL 9000 console's architecture, it is highly integrated with the console's other features, enabling streamlined workflow and consistent sound quality.
Characteristic SSL Sound: The compressor is known for imparting the classic SSL "glue" and punch, especially when used on drums, vocals, and mix buses, contributing to the iconic SSL sound.
High-Quality Build and Components: Like other SSL modules, it is built with professional-grade components, ensuring durability and reliability in studio environments.

Overall, the SSL 9000 Channel Compressor is celebrated for its combination of transparency, versatility, and the distinctive SSL sonic character, making it a favored choice for professional mixing engineers.

Soul G Buss. Emulates the SSL 9000 G Bus Compressor.

The SSL 9000 G Bus Compressor is renowned for its distinctive sound and musical characteristics, making it a highly sought-after hardware compressor in professional recording and mixing. It is best known for:
Musical Compression: The SSL G Bus Compressor imparts a cohesive, glue-like quality to mixes, helping elements sit together smoothly and dynamically.
Punch and Clarity: It enhances punch and clarity without overly squashing the dynamics, preserving transients while controlling peaks.
Signature SSL Sound: It contributes the classic SSL "glue" effect, characterized by a musical and transparent compression that has become a hallmark of many hit recordings.
Versatility: While primarily used on the stereo mix bus, it is also effective on individual tracks such as drums, vocals, and bass.
Historical Significance: The SSL 9000 G Bus Compressor has been a staple in professional studios since the 1980s, appreciated for its reliability and consistent sonic performance.

In summary, the SSL 9000 G Bus Compressor is celebrated for its ability to enhance mixes with a musical, transparent compression that adds cohesiveness, punch, and clarity—traits that have made it a classic and enduring piece of studio gear.

Even Compressor/Lim. Emulates: Neve 33609.

The Neve 33609 is renowned for its high-quality mono channel strip, widely regarded as a flagship preamp and dynamics module from Neve's classic 80-series consoles. It is particularly celebrated for its rich, musical sound, characterized by warm, punchy preamplification and smooth, musical compression.

The 33609 is highly sought after in professional recording studios for its ability to add character and depth to vocals, drums, and other instruments, making it a favorite among engineers and producers seeking the legendary Neve sound.
Some of its key features are:

Compressor and Limiter Functions: Combines compression and limiting capabilities to control dynamic range effectively.
Variable Attack and Release: Adjustable attack and release times for precise dynamic shaping.
Threshold Control: Sets the level at which compression begins.

Ratio Control: Adjustable compression ratio to determine the degree of compression. Make-Up Gain: Allows compensation for level reduction after compression.

Sidechain Filter: Optional high-pass filter in the sidechain to prevent low-frequency buildup from affecting compression.
High-Resolution Metering: VU or peak meters to monitor gain reduction and output levels accurately.
Classic Neve Sound: Known for its musical, warm compression characteristic favored in professional recording and mixing.

Applications:
Tracking (vocals, drums, instruments), Mix Buss Compression, Mastering, Broadcast

Eternal Bliss. Emulates the Elysia Mpressor.

Elysia Mpressor is known for its innovative approach to music production, particularly in blending genres such as electronic, experimental, and ambient sounds. It has gained recognition for its unique sound design, creative use of synthesizers, and compelling compositions that often explore themes of emotion and atmosphere. The original is a top-end unit with pristine electronics and several unique features.
The Elysia Mpressor is a high-end stereo compressor known for its transparent and musical dynamics control. The Elysia Mpressor is often praised for its transparency, musicality, and versatility in mastering and mix bus applications. Some of its key features include:
Pure VCA Compression: Utilizes a proprietary VCA design for precise and clean compression, allowing for transparent dynamics control.
High-Quality Audio Path: Designed with high-grade components to ensure pristine audio fidelity and minimal coloration.

Flexible Control Parameters:

Threshold: Adjustable to set the level at which compression begins. Ratio: Multiple ratio settings for gentle to aggressive compression.

Attack and Release Times: Precise control for shaping the compression response. Knee Control: Allows for soft or hard knee compression characteristics.
Make-up Gain: Compensates for level reduction caused by compression.

Stereo Linking: Ensures balanced processing of stereo signals for coherent sound. Metering:
Gain Reduction Meter: Visual indicator of compression amount. Input/Output Meters: Monitor signal levels throughout the process.
Side-Chain Filtering: Optional high-pass filter in the side-chain to prevent bass frequencies from triggering compression excessively.
Bypass Function: Allows quick A/B comparison between compressed and uncompressed signals.
Auto-Fast button: Allows for manual setting of the attack timing yet kicks in with faster attack timing when it detects faster and louder signals.
Anti-Log release button: Inverts typical release timing providing long releases with high gain reduction and fast release times with low gain reduction.
Negative Ratios: Negative ratios don't limit the volume to the threshold but turn the volume down below the threshold. As the input goes up, the volume reduction increases.
Gain Reduction Limiter: This prevents gain reduction from exceeding the specified amount no matter how loud the input, low the threshold or how high the ratio.

76 Limiter Amp. Emulates: UREI/Universal Audio 1176 FET Compressor.

The UREI/Universal Audio 1176 FET Compressor is renowned for its distinctive sound and versatile performance in professional audio production. Introduced in the late 1960s, it became one of the most popular and iconic hardware compressors in recording studios worldwide. Some of its key features include:
Fast Attack and Release Times: The 1176 is celebrated for its extremely rapid attack and release, enabling it to effectively tame transient peaks and add punch to drums, vocals, and other instruments.
Distinctive "All-Button" Mode: Engaging all ratio buttons creates a unique, aggressive compression characteristic often used for special effects or more pronounced compression.
FET (Field Effect Transistor) Circuitry: Its design utilizes FETs to emulate classic optical or variable-mu compressors, but with the added benefit of faster response times and a distinctive sonic character.
Coloration and Harmonics: The 1176 imparts a recognizable sonic flavor, adding warmth, presence, and a certain "glue" to mixes, which many engineers seek out.
Versatility: It can be used on a variety of sources—vocals, drums, bass, and even stereo bus compression—making it a staple in mixing and mastering.

Historical Significance: The 1176 is considered a classic piece of gear that helped shape the sound of modern music from rock and pop to jazz and beyond.

In summary the UREI/Universal Audio 1176 FET Compressor is known for its fast, aggressive compression, distinctive tonal coloration, and versatility, making it a favorite among engineers and producers for adding punch, character, and consistency to recordings and is fantastic on drums, vocals, bass, and parallel compression. Always remember that the timing knobs are reversed from normal with 1 being the slowest and 7 being the fastest.

LA Leveler. Emulates the Teletronix LA-2A.

The Teletronix LA-2A is renowned for its smooth, musical compression characteristics. It is a classic optical leveling amplifier that uses an electro-optical process to achieve its compression, resulting in a natural and transparent sound.

The LA-2A is especially favored in recording and mixing for vocals, bass, and other instruments where gentle, transparent compression is desired. Its distinctive sound and ease of use have made it a legendary piece of studio equipment in the world of professional audio. The LA-2A remains a favorite among audio engineers and producers for its musicality and classic tone, making it a staple piece of equipment in many professional and home studios. Some of its key features include:

Optical Gain Reduction: Utilizes an electro-luminescent panel and a photo-sensor to achieve natural,Program-dependent compression with minimal distortion.
Vintage Tube Design: Incorporates a tube-based amplifier (typically a 12AX7 tube), contributing to its warm, musical sound.
Simple Control Interface:

Peak Reduction (Compression) knob: Adjusts the amount of gain reduction.

Gain (Make-up Gain) knob: Compensates for the gain reduction to preserve output level.
Peak Reduction Meter: Displays the amount of compression applied. Input Level Meter: Shows incoming signal level.
Automatic Leveling: Its design provides a natural, transparent compression ideal for vocals, bass, and other instruments requiring gentle leveling.
Slow Attack and Release: The LA-2A features inherently slow attack and release times, making it excellent for smoothing out dynamic signals without introducing artifacts.
Classic Sound: Its unique optical compression circuit imparts a warm, rounded tone that's highly sought-after in mixing and mastering.
Side-Chain Capabilities: While primarily an automatic leveler, it lacks advanced side-chain filtering but excels at transparent compression.

Fairkid Model 670. Emulates: Fairchild 670.

The Fairchild 670 is renowned for being one of the most legendary and highly regarded vintage compressor/limiter units in audio production. Manufactured by Fairchild Recording Equipment Corporation in the 1950s and 1960s, it is particularly famous for its exceptional sound quality, warm compression characteristics, and its distinctive, smooth gain reduction.

The 670 is often considered a benchmark for audio compression, especially in high-end recording studios, and is prized by engineers and producers for its ability to add a rich, musical character to vocals, drums, and other instruments.

Due to its rarity and exceptional performance, the Fairchild 670 has become a highly sought-after piece of vintage audio gear, symbolizing the golden age of analog recording equipment.

The Fairchild 670 is one of the most legendary and sought-after vintage audio compressors, renowned for its warm, musical compression and its use in top studios around the world. Here are some key features of the Fairchild 670:

Stereo Compression: Designed for stereo operation, providing dual-channel compression within a single unit, ensuring consistent stereo imaging.

High Power and Large Size: Extremely heavy and large, often weighing over 400 pounds, reflecting its robust construction and powerful circuitry.
Opto-Triode Gain Reduction: Employs opto-electronic gain reduction elements for smooth, musical compression.
Attack and Release Controls: Features adjustable attack and release times, allowing precise tailoring of the compression response.
Threshold and Ratio Controls: Provides control over the level at which compression begins and the degree of compression applied.
Output Gain Control: Allows for makeup gain to compensate for level reduction caused by compression.
Variable Compression Ratios: Offers a range of ratios, typically up to 20:1, for subtle or aggressive compression.
High-Fidelity Audio Path: Designed to preserve audio quality with minimal coloration, enhancing warmth and clarity.
Vintage and Rare: Due to its scarcity and legendary status, original units are highly valued and often considered collector's items.
Additional Notes: The Fairchild 670 is prized not only for its technical capabilities but also for its unique sonic signature—often described as warm, smooth, and musical. It's widely used in mastering, tracking, and mixing for vocals, drums, and other critical elements.

No Stressor. Emulates the Emperical Labs EL8 Distressor.

The Empirical Labs EL8 Distressor is renowned for its versatility and high-quality compression capabilities. It is a widely used hardware compressor known for its ability to emulate a variety of classic compression styles while offering unique features that make it suitable for a broad range of audio applications. Some of its key characteristics include:
Versatility: The EL8 Distressor can function as a gentle opto compressor or as a more aggressive, “in-your-face” compressor, making it suitable for vocals, drums, bass, guitars, and even mix bus compression.
Distinctive Sound: It is prized for its ability to add character and “glue” to mixes, often described as bringing a sense of punch and cohesion to recordings.
Unique Features: The unit includes controls for “Distortion,” “Nuke” (extreme compression), and “Grain” modes, allowing users to creatively shape their sound.
Emulation of Classic Compressors: It can emulate the behavior of classic units like the 1176 and LA-2A, providing a range of compression styles from transparent to heavily colored.
Musical and Transparent Options: It offers both transparent compression and more aggressive, colored compression, giving engineers creative flexibility.

Overall, the Empirical Labs EL8 Distressor is known for its flexibility, distinctive tone, and its ability to serve as both a transparent and an aggressively characterful compressor, making it a staple in professional studios worldwide.

Controls & Special Settings:

Ratio Adjustment: Controls ratio, knee and threshold settings. 3:1 and lower: Are soft knee.
4:1 and 6:1: Are soft knee below the threshold and move to hard knee above the threshold.
10:1: Is a special optical type compression. Additional Notes/Features
20:1: Attack 10 Release 0 emulates an LA-2A. Nuke is a brickwall limiter.
Input drives the signal into the threshold.

Setting 5-5-5-5 on all four knobs is the classic starting point.

The Distressor Manual is available at https://www.empiricallabs.com/distressor

PIA2250 Rack. Emulates: API 225L 200 series module.

The API 225L 200 Series module is known for its high-quality, versatile analog signal processing capabilities, particularly in professional audio and broadcast applications. It is part of API's 200 Series channel modules, which are renowned for their rugged construction, transparent sound, and classic API tone.

Specifically, the 225L module is often used for microphone preamplification, line-level amplification, or as a versatile gain stage, providing clean, warm, and detailed audio signals with precise control. Its reputation stems from its consistent performance, durability, and the signature API sound that is highly valued in recording studios, broadcast environments, and sound reinforcement systems.

The API 225L Compressor is ideal for all studio, live sound and broadcast applications. Regardless of the threshold or ratio settings, the output level always remains at unity. This unique feature allows for real-time adjustments without the need for changing the output level.

Release time is adjusted by rotating the REL knob. Release time constants: 50m/sec to 3/sec. Attack time is switch selectable to Fast (2mS), Medium (18mS) or Slow (75mS). The 225L is designed for individual channel use or, through the use of the external LINK

function, two units can be combined for Stereo applications via a rear access pin. The 225L also has a side-chain input for the detector amplifier.
The 225L can be used in the Legacy, Legacy Plus or Vision Series Consoles, or in the L200 Rack. The 225L Compressor makes use of the 2520 and 2510 op-amps and therefore exhibits the reliability, long life and uniformity which are characteristic of all API products.
Website: https://www.odysseyprosound.com/dynamics/api-225l

LTA100 Leveler. Emulates the Summit Audio TLA-100.

The Summit Audio TLA-100 is renowned for its warm, transparent, and musical tube-based compression. It is widely appreciated in both music production and mixing for its ability to add glue and cohesion to tracks while preserving the natural dynamics of the source.

The TLA-100 is particularly favored for vocals, bass, and acoustic instruments, thanks to its smooth compression characteristics and the characteristic harmonic richness imparted by its tube circuitry. Its intuitive controls and high-quality build make it a popular choice among engineers seeking a classic, analog-style compressor.

The Summit TLA-100 is appreciated for its musicality and ability to add warmth and punch to recordings, making it a staple in many professional studios. Some of the key features are:

Compression Characteristics:

Variable attack, release, and ratio controls for precise tailoring.

Soft knee compression for smooth control.

High-pass sidechain filter to prevent bass frequencies from triggering compression excessively.
Controls and Parameters:

Threshold control for setting the compression onset. Ratio control for adjusting the compression amount. Attack and release knobs for dynamic response shaping. Makeup gain for compensating gain reduction.
Metering:

VU meter displaying gain reduction.

Optional peak indicator for transient handling. Audio Quality and Features:
Warm, musical compression characteristic.

Suitable for vocals, drums, bass, and mix bus applications. Sidechain filtering for more transparent compression.
Additional Features:

Bypass switch for A/B comparison.

The Summit Audio TLA-100 PDF Manual is available at https://www.summitaudio.com/manual.php?m=tla-100a

Wave Designer. Emulates the SPL Transient Designer.

The SPL Transient Designer is known for its innovative ability to shape and manipulate the transients of audio signals. Specifically, it allows users to independently adjust the attack and sustain portions of a sound without affecting the overall level or tone.

This makes it a powerful tool for enhancing punchiness in drums, tightening up sounds, or softening transients to create a more controlled and polished mix. Its intuitive interface and real-time processing have made it a popular choice among audio engineers and producers for dynamic sound shaping and transient management.

The SPL Transient Designer is a popular audio processing tool used to shape the attack and sustain characteristics of audio signals, particularly drums and percussion. Here are some of its key features:

Transient Enhancement and Reduction:
Allows users to either emphasize or reduce the attack (initial hit) and sustain (tail) of the sound.
Simple, Intuitive Controls:

Attack: Adjusts the initial transient's strength. Sustain: Controls the tail length and decay.

Output Gain: Controls the amount of output makeup gain.

Dynamic Processing: Unlike traditional compressors or EQs, it specifically targets the transient portion of the audio, making it highly effective for drums, percussion, and plucked instruments.
Real-Time Processing: Provides immediate results, useful for live mixing or quick edits. Versatile Use Cases: Tightening drums for a punchier sound. Loosening instruments for a more natural feel. Creative effects by drastically altering transients.
Minimal Phase Distortion: Designed to process transients without introducing significant phase issues, preserving the natural sound.
Additional Features: Bypass options for quick A/B comparisons.

PSE LA Combo. Emulates the Vintage LA-style Compressors.

The PSE LA Combo Compressor emulates the classic sound and characteristics of vintage LA-style compressors, specifically those associated with large-format analog mixing consoles like the Neve and SSL desks. It aims to recreate the warm, musical compression and punchy dynamics that are typical of analog hardware used in professional recording and mixing environments.

The plugin often combines multiple compression stages or circuitry to capture the richness, glue, and subtle coloration that analog LA-style compressors are known for, making it a popular choice for adding warmth and cohesion to vocals, drums, and full mixes.

The PSE LA Combo Compressor is known for its distinctive combination of compression and limiting capabilities, often appreciated in audio production for its versatility and unique tonal characteristics. Specifically, it is renowned for:

Versatility: It can serve as both a compressor and a limiter, making it suitable for a wide range of audio processing tasks.

Unique Sound Character: The PSE LA Combo is celebrated for its warm, musical compression that adds character and punch to vocals, drums, and other instruments.

Ease of Use: With intuitive controls, it allows engineers and producers to quickly shape their sound without complex setup.

Overall, the PSE LA Combo Compressor is known for its ability to enhance tracks with smooth, musical compression while adding a touch of harmonic richness, making it a favorite among audio professionals seeking a vintage vibe.

Auto Rider. Emulates the Waves Vocal Rider.

The Waves Vocal Rider is known for its ability to automatically and transparently ride vocal levels in a mix. It dynamically adjusts the volume of vocal tracks in real-time, ensuring consistent presence and clarity without the need for manual fader riding. This tool is widely used in mixing to streamline the process of achieving balanced vocals, saving time and maintaining a natural sound.

Pre FX Effects Section REVERBS

Hall Reverb

A hall reverb is a type of audio effect that simulates the reverberation characteristics of a large concert hall or auditorium. It is used in music production, sound design, and audio engineering to create a sense of space and depth in a sound recording or live performance.
How It Works:

When a sound is produced in a large space like a hall, it reflects off walls, ceilings, and other surfaces. These reflections arrive at the listener's ears at different times and intensities, creating a complex, spacious sound.
Hall reverb mimics these reflections digitally or through hardware, adding a natural echo and spaciousness to the original sound.
Characteristics of Hall Reverb:

Long Decay Time: The reverberation lasts longer, often several seconds, giving a sense of grandeur and openness.
Smooth, Diffused Reflections: The reflections blend smoothly, avoiding harsh echoes.

Lush and Spacious Sound: It enhances vocals, orchestral instruments, and other sources that benefit from a grand, ambient feel.
Applications:

Making vocals sound more expansive and natural. Adding depth to orchestral recordings.
Creating an atmospheric background in various music genres. Historical significance:
One notable group that used a Hall Reverb in their music is The Beatles. They employed a Hall Reverb to create spacious, ambient sounds in several of their recordings. For example, their song "A Day in the Life" features prominent use of a large hall reverb to give it an expansive, atmospheric quality.
Other artists and groups across different genres have also utilized Hall Reverb to add depth and space to their recordings, but The Beatles are among the most historically significant and well-known for pioneering the use of this effect in popular music.
In summary, a hall reverb is a reverb effect that emulates the acoustic environment of a large hall, enriching the sound with a sense of space and ambiance.

Room Reverb

A room reverb, or room reverberation, refers to the natural or artificial echo and spatial reflections that occur when sound waves bounce off surfaces within a room or enclosed space. It creates the sense of space and depth in audio recordings or live sound by simulating how sound behaves in different environments.
In audio processing, room reverb is often added using reverb effects to make a sound

feel like it’s happening in a particular type of room—such as a small studio, a cathedral, or a concert hall—enhancing realism and emotional impact. It can be customized with parameters like decay time, size, damping, and early reflections to match the desired acoustic environment.

Characteristics of Room Reverb:

Short to Moderate Decay Time. The reverberation in a room reverb tends to decay relatively quickly, usually within a fraction of a second to a couple of seconds. This short decay helps maintain clarity while adding a sense of space.
Reflections with Less Diffusion:

In a typical room, reflections are more direct and less diffused compared to larger spaces. The reflections often arrive sooner and are more defined, creating a sense of intimacy.
Intimate and Natural Sound:

Room reverb provides a natural, close-mic’d ambiance, making it suitable for capturing the original character of the sound source. It enhances the sense of being in a small, enclosed space.
Frequency Response:

Often characterized by a relatively flat or slightly colored frequency response, depending on the room’s surfaces. Some room reverbs emphasize certain frequencies based on the room’s acoustics.

Applications:
Emphasizing the natural sound of vocals or instruments in a small space. Adding subtle ambiance without overwhelming the original sound.
Creating a realistic sense of intimacy or closeness in recordings. Historical significance:
Several musical groups and artists have used room reverb or natural room acoustics as a significant element in their recordings to create a sense of space and atmosphere.
One notable example is:

The Beatles – In their song "A Day in the Life", especially during the orchestral crescendos and final piano chord, natural room reverb and ambient effects are used to enhance the spacious sound. Additionally, the use of room reverb can be heard in various tracks throughout their catalog, often achieved through techniques like recording in large rooms or using artificial reverb to emulate natural spaces.
Other notable examples include:

Pink Floyd – Known for their expansive soundscapes, many of their tracks (such as
"Echoes" or "Shine On You Crazy Diamond") incorporate room reverb, either naturally or artificially, to create immersive atmospheres & Radiohead – In songs like "Everything in Its Right Place," the production employs room reverb to add depth and space to their electronic and experimental sounds.

Brian Eno – As a pioneer of ambient music, Eno frequently used room reverb and
natural space recordings to craft lush, immersive sound environments. In general, many artists and producers have used room reverb either through recording in large spaces, using reverb plates and chambers, or adding artificial reverb effects during mixing to achieve desired spatial qualities in their songs.

Summary: Room reverb mimics the acoustic environment of a small to medium-sized space, characterized by shorter decay times, distinct and less diffused reflections, and an intimate, natural quality. It’s often used to

add a realistic sense of space while maintaining clarity and detail. Basically, room reverb is both a natural acoustic phenomenon and an audio effect used to add spatial characteristics to sound.

Chamber Reverb

A Chamber Reverb is a type of artificial reverberation effect that simulates the sound characteristics of a small, enclosed space—originally created using a dedicated sound chamber or echo chamber. In the context of audio production, it typically refers to a reverb effect that emulates the acoustics of a vintage or small room environment, often characterized by a warm, smooth, and natural decay.
Historically, chamber reverb was produced by sending audio signals into a specially designed physical space—such as a tiled or echo chamber—and capturing the reflected sound via microphones. In modern digital audio production, chamber reverb effects are often emulated through software plugins that model the acoustic properties of these spaces, offering a distinct, lush reverb sound that adds depth and character to vocals, instruments, and mixes.
Key characteristics of Chamber Reverb: Mimics small, enclosed spaces.
Offers a warm, natural ambiance.

Often used to add subtle depth without overwhelming the original sound.

Can be more musical and less artificial compared to algorithmic or hall reverb settings. Historical significance: Many artists and groups have used chamber reverb to achieve a lush, spacious sound in their recordings. One notable example is The Beatles, who frequently employed chamber reverb—most famously on songs like "A Day in the Life" and "Tomorrow Never Knows." They used the "Leslie" chamber (a specially designed reverberation chamber) at Abbey Road Studios to create their distinctive sound.
Another example is Pink Floyd, who used various types of reverb, including chamber reverb, to craft atmospheric textures in their music. Additionally, artists like Simon & Garfunkel and The Beach Boys incorporated chamber reverb techniques in their recordings for a spacious, immersive effect.
Overall, chamber reverb is valued for its ability to create a vintage, intimate, and organic reverb sound that enhances the spatial quality of recorded audio.

Plate Reverb

A Plate Reverb is a type of artificial reverberation device that uses a large metal plate to produce reverberation effects. It was widely used in recording studios before digital reverb technologies became prevalent.
How it works:

A transducer (similar to a loudspeaker) is attached to the metal plate to convert an audio signal into vibrations.
These vibrations travel across the metal surface, reflecting and dispersing.

A pickup (like a microphone) mounted on or near the plate captures the vibrations, converting them back into an audio signal that contains the reverb effect.
Characteristics:

Produces a dense, smooth, and warm reverb sound. Has a distinctive metallic, resonant quality.

Offers a relatively short decay time compared to natural reverberation, but can be adjusted with damping and size.
Historical significance:

Plate reverbs were popular in the 1950s and 1960s for vocals, drums, and other instruments. Famous examples include the EMT 140, one of the most iconic plate reverberators. Today, digital and software reverb plugins often emulate the sound of plate reverbs, allowing for easier use without the need for physical equipment.
Several artists and groups have used plate reverbs to create distinctive sounds in their recordings. One notable example is The Beatles, who famously used a plate reverb on John Lennon's vocals and other instruments during the production of songs like "A Day in the Life" and "Tomorrow Never Knows."
Another prominent example is Pink Floyd, who employed plate reverbs extensively in their recordings to achieve lush, spacious soundscapes, especially in albums like The Dark Side of the Moon.
Led Zeppelin also used plate reverbs in their recordings, notably on Robert Plant's vocals to add depth and resonance. Additionally, The Beach Boys and The Rolling Stones are known to have utilized plate reverbs in their studio work to enhance their recordings.

Concert Reverb

A Concert Reverb refers to the natural or artificially created reverberation experienced during live musical performances in concert halls or large venues.
Reverberation (or reverb) is the persistence of sound after the original source has stopped, caused by reflections of sound waves bouncing off walls, ceilings, and other surfaces.
In the context of a concert, "concert reverb" often describes the characteristic acoustic qualities of a specific venue—how it amplifies, sustains, and diffuses sound—contributing to the overall ambiance and perceived space of the performance. For example:
Natural concert reverb: The inherent echo and reverberation created by a venue's architecture.
Artificial or added reverb: Signal processing effects used in sound reinforcement systems to emulate or enhance the natural reverb characteristics of a venue.
Understanding concert reverb is essential for sound engineers, musicians, and producers to achieve the desired spatial and acoustic effects, whether in live sound

reinforcement or in studio recordings aiming to replicate the feeling of a live concert environment.
Historical significance:

The band Pink Floyd is notably associated with the use of "concert reverb" effects in their music. One prominent example is their song "Echoes" from the album Meddle (1971), where they employed extensive reverb and echo effects to create spacious, atmospheric soundscapes.

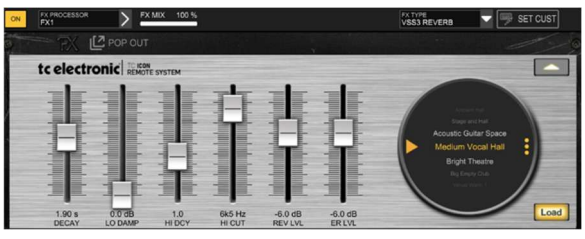
Additionally, various progressive and psychedelic rock bands during the late 1960s and early 1970s experimented with concert reverb and similar effects to craft immersive auditory experiences, though Pink Floyd remains one of the most recognized for their innovative use of such effects.

Ambience Reverb

An Ambience Reverb is a type of reverb effect used in audio production to create a sense of space and environment around a sound. It simulates the natural reflections and echoes that occur in a physical space, such as a room, hall, or outdoor environment, adding depth and dimension to the audio.
Unlike more prominent reverb effects that might produce noticeable echoes or long decay times, ambience reverb typically features a subtle, gentle reverberation with a relatively short decay. Its purpose is to enhance the overall atmosphere without overpowering the original sound, making it ideal for creating a natural, immersive soundscape.
Key characteristics of Ambience Reverb: Short to medium decay time.
Low to moderate reverb level. Designed to add spatial context subtly.

Often used on vocals, instruments, or entire mixes to create a sense of space. Historical significance:
Several musical groups and artists have used ambient reverb effects to create atmospheric sounds in their songs. One notable example is Pink Floyd, who frequently employed expansive reverb and ambient effects to craft their signature psychedelic and progressive rock sound.
For instance, their song "Echoes" features extensive use of reverb and ambient textures to create a spacious, immersive atmosphere.
Another example is U2, particularly in their song "Where the Streets Have No Name," where reverb and ambient effects are used to enhance the song's expansive feel.
Additionally, artists like Brian Eno—known as a pioneer of ambient music—used ambient reverb extensively in his compositions to generate lush, atmospheric soundscapes.
In summary: An Ambience Reverb enriches a sound by mimicking the natural acoustic environment, providing a lush, spacious background that helps elements sit better within a mix.

VSS3 Reverb



VSS3 Reverb is a digital reverberation algorithm developed by Yamaha, originally introduced as part of their professional digital mixing consoles and effects processors. It is renowned for its high-quality, natural-sounding reverberation and is widely used in music production, live sound, and post-production. It has 109 adjustable reverbs!
Key features of VSS3 Reverb include:

Algorithm Type: It is a form of algorithmic reverb, designed to simulate the reflections and decay of sound in a space.
Sound Quality: Known for its smooth, natural reverberation with a rich tail and spacious ambience.

Control Parameters: Typically offers controls for parameters such as decay time, early reflections, diffusion, density, and damping, allowing users to tailor the reverb to suit various acoustic environments.
Usage: Frequently integrated into Yamaha's digital consoles and effects units, but also emulated or included in third-party plugins and hardware.

In summary: The VSS3 Reverb is a highly regarded digital reverb algorithm that provides realistic and musical reverberation effects, making it a popular choice among audio engineers and producers for enhancing the spatial quality of recordings and live sound.

Vintage Room Reverb

A Vintage Room Reverb refers to a type of reverb effect that emulates the acoustic characteristics of classic, older recording spaces or analog reverb units from past decades. It aims to recreate the warm, lush, and sometimes slightly colored reverberation that was typical of vintage hardware or famous recording studios.
Key characteristics of Vintage Room Reverb include:

Analog Warmth: Often featuring subtle saturation and harmonic distortion reminiscent of analog equipment.
Distinct Acoustic Spaces: Mimicking the unique reverberation qualities of historic rooms, such as concert halls, studios, or chambers used in classic recordings.
Emulation of Vintage Hardware: Many plugins or hardware units designed as Vintage Room Reverbs emulate classic spring, plate, or chamber reverbs from mid-20th century equipment.
These reverbs are popular in music production for adding a sense of nostalgia, richness, and character to recordings, especially when aiming for a vintage or retro sound aesthetic.

Historical significance:

Several musical groups and artists have used vintage room reverb to create distinctive sounds in their recordings. One notable example is The Beatles, who utilized vintage reverb effects, including spring and plate reverbs, to craft their iconic sound. For instance, the song "A Day in the Life" features the use of vintage reverb techniques to add depth and atmosphere.
Another example is Pink Floyd, who extensively used vintage reverb units like the EMT 140 plate reverb to produce spacious and immersive soundscapes in tracks such as "Time" and "Echoes."
Additionally, artists like The Beach Boys and Led Zeppelin incorporated vintage reverb effects to achieve their signature sounds, often employing classic spring reverbs and early digital units that emulate vintage characteristics.

Vintage Reverb

A Vintage Reverb refers to a type of audio reverb effect that emulates the sound characteristics of classic reverb units and techniques from earlier eras, typically from the 1950s to the 1980s. These reverb sounds are often prized for their warm, lush, and sometimes distinctively colored or saturated quality, which can add a nostalgic or timeless character to recordings.
Common examples of vintage reverb units include:

Plate Reverbs: Like the EMT 140, known for their smooth, dense reverberation.

Spring Reverbs: Found in vintage guitar amplifiers, offering a distinctive metallic and boingy sound.
Spring Reverb Units: External hardware units used in studios or on instruments.

Hardware Spring Reverb Units: Classic outboard gear that uses actual springs to create reverb effects.

In addition to hardware units, "vintage reverb" can also refer to software plugins modeled after these classic units, capturing their unique tonal qualities and saturation characteristics.
Characteristics of Vintage Reverb:

Warmth and musicality due to analog circuitry and saturation. Distinctive coloration and character.
Often more subtle and less pristine than modern digital reverbs. Can evoke a sense of nostalgia or classic vibe in recordings.
Historical significance:

Many bands and artists across various genres have used vintage reverb units to achieve their distinctive sounds. One notable example is The Beach Boys. They famously used an EMT 140 plate reverb on many of their recordings, including the iconic "Good Vibrations," to create lush, spacious sounds.
Other artists and groups that have utilized vintage reverb units include:

The Beatles – Used various vintage reverb units, including the EMT 140, for their experimental and atmospheric soundscapes.
Pink Floyd – Employed vintage reverbs like the EMT 140 and EMT 250 to craft their atmospheric textures.
U2 – Used vintage plate reverbs during the 1980s for certain tracks, such as "With or Without You."
The Rolling Stones – Utilized vintage reverbs for their recordings in the 1960s and 70s.

Using vintage reverb can add a special personality to recordings, making them sound more organic, spacious, and emotionally engaging.

Vintage Plate

A Vintage Plate Reverb is an early type of artificial reverb that uses a large, suspended metal plate to produce reverberation effects. Invented in the mid-20th century, especially popular from the 1950s through the 1970s, plate reverb units consist of a thin, circular or rectangular metal plate that vibrates when an audio signal is fed into it via transducers (pickups).
The vibrations are then captured by pickups placed on or near the plate, converting them back into an audio signal with a lush, dense reverberation characteristic.
Key features of Vintage Plate Reverb:

Distinct sound: Known for its warm, smooth, and dense reverb with a slightly metallic quality.
Physical construction: Uses a large metal plate, a transducer (input), pickups (output), and mechanical components.

Historical significance:

One notable group that used a vintage plate reverb is Pink Floyd. They famously employed a classic EMT 140 plate reverb on several of their recordings, including the iconic track "Shine On You Crazy Diamond." The EMT 140 plate reverb contributed to the spacious, ethereal sound characteristic of Pink Floyd's atmospheric style.
Other artists and producers from the 1960s and 1970s also used vintage plate reverbs, such as The Beatles and The Beach Boys, to achieve lush, immersive reverberation effects in their recordings. It is widely used in recording studios before digital reverbs, favored for vocals, drums, and instruments.
Examples: The EMT 140 as seen below is one of the most iconic vintage plate reverb units ever developed.

Today, while digital emulations and plugins replicate the sound, vintage plate reverbs are prized for their unique sonic qualities and nostalgic appeal.

Blue Plate

A Blue Plate Reverb is a classic, plate-type reverberation unit that was originally manufactured by the American company Ampex in the 1950s. It is renowned for its distinctive, warm, and smooth reverb sound, which became popular in recording studios for adding depth and ambiance to vocals, drums, and other instruments.
Key features of the Blue Plate Reverb include:

Design: It uses a large, thin steel or aluminum plate suspended within a metal frame, which vibrates to produce reverb. An electromagnetic transducer (speaker) sends audio signals to the plate, and pickups (similar to those in electric guitars) capture the vibrations as reverberated sound.
Sound Characteristics: The Blue Plate Reverb is known for its rich, musical decay with a slightly darker and more natural quality compared to digital reverbs. Its sound is often described as warm, smooth, and slightly elongated.
Historical Significance:

It was widely used in the 1950s and 1960s recording industry, notably in jazz, rock, and pop music. Its distinctive sound has made it a sought-after piece of vintage studio gear. The Blue Plate Reverb, a classic and sought-after piece of studio hardware, has been used by numerous artists across various genres.
One notable example is U2, who employed the Blue Plate Reverb during the recording of their album The Joshua Tree (1987). It contributed to the distinctive spacious and atmospheric sound characteristic of some tracks on that album.
Additionally, The Rolling Stones are known to have used the Blue Plate Reverb in their recordings, notably during the sessions for Some Girls (1978), to achieve a particular vintage reverb sound.
While these are some prominent examples, many producers and engineers have favored the Blue Plate Reverb for its rich, warm, and musical reverb qualities, making it a popular choice for various recording projects.
Contemporary Usage:
Today, the Blue Plate Reverb is considered a vintage or boutique piece of gear. Many modern reverb plugins emulate its characteristics, and original units are prized by collectors and engineers seeking its classic tonal qualities.
Summary: The Blue Plate Reverb is a vintage plate reverb unit known for its warm, musical reverb sound, historically used in professional recording studios to add depth and character to recordings.

Gated Reverb

A Gated Reverb is a popular audio effect used in music production, especially prominent in the 1980s. It combines a reverberation (reverb) with a noise gate to create a distinctive, punchy sound.
How it works:

Reverb Tail: First, a sound (often a drum hit or vocal) is processed with a reverb to create a spacious, echoing tail.
Gating: A noise gate is then applied to the reverb tail, allowing only the initial part of the reverberation to pass through before abruptly cutting it off.
Result: The effect produces a sharp, sudden decay rather than a lingering reverb, giving a " gated" or "cut-off" sound. This creates a punchy, energetic effect that emphasizes percussive hits.
Historical Significance:

The gated reverb sound became iconic with songs like Phil Collins' "In the Air Tonight" and other 80s productions. It was achieved by combining traditional reverb with a noise gate, and often involved additional processing to shape the sound.
In summary: A Gated Reverb is an audio effect where a reverb's decay is abruptly cut off by a noise gate, resulting in a distinctive, punchy reverberation characteristic of many 80s hits.

Reverse Reverb

A Reverse Reverb is an audio effect where the reverb tail of a sound is played backward, creating a swelling or swelling-like sound that leads into the original audio. Instead of the reverb trailing after the initial sound, the reversed reverb builds up before the note or phrase, producing a unique, ethereal, and often haunting effect.
How it works:

The original sound (e.g., a vocal or instrument) is recorded or processed. The reverb (or the entire sound with reverb applied) is reversed in time.
The reversed reverb is then blended back with the original sound, so the swelling appears to lead into the sound.
Often, the reversed reverb is trimmed or manipulated to fit the desired effect. Uses:

Creating tension or anticipation in music. Adding an otherworldly or surreal atmosphere.
Emphasizing transitions or emphasizing certain sounds. Historical Significance:
Pink Floyd employed reverse reverb effects in their music. One notable example is in the song "Echoes" from their 1971 album Meddle. The band used various studio techniques, including reverse reverb, to create atmospheric textures and swirling sounds that contribute to the song's immersive, spacious feel.
The reverse reverb helped produce the shimmering, other worldly effects that are characteristic of Pink Floyd's experimental approach during that period. Additionally, Pink Floyd was known for their innovative studio effects and ambient soundscapes, often utilizing reverse reverb and other tape manipulation techniques to craft their distinctive sound.
In summary: Reverse Reverb is a creative audio effect that reverses the reverb tail to produce a swelling sound that prefaces the original audio, resulting in a distinctive, dreamy, or haunting sonic texture.

Delay Reverb

A Delay Reverb is an audio effect that combines delayed sound signals with reverberation to create a spacious and immersive soundscape. Essentially, it involves adding echoes (delays) to the reverberated sound, resulting in a sense of depth and movement in the audio.
How it works:

Reverb simulates the natural reflections of sound in a space, creating a sense of environment.
Delay involves repeating the sound after a set period, producing echo-like effects.

When combined, a delay reverb can produce complex textures, such as cascading echoes within a reverberant space, or a sense of grandeur and width in the sound. This effect is popular in music production, sound design, and live performance to add dimension and atmosphere.

Historical Significance:

One notable example of a group that used a delayed reverb effect in their music was the British band “The Beatles”. Specifically, they employed a technique called "double tracking" combined with reverb and delay effects to create spacious, lush sounds on many of their recordings.
A prominent track where you can hear a delayed reverb effect is "Tomorrow Never Knows" from the album Revolver (1966). The song features innovative studio techniques, including tape delay and echo effects that contribute to its psychedelic soundscape.
Another example is Pink Floyd, who extensively used delay and reverb effects in their atmospheric and spacey soundscapes, especially in albums like The Dark Side of the Moon and Wish You Were Here.
In summary: A Delay Reverb is an audio effect that layers delayed echoes onto reverberation, enhancing spatial and temporal qualities of the sound.

Shimmer Reverb

A Shimmer Reverb is a type of audio effect that combines traditional reverb with
pitch-shifting elements to create a lush, ethereal, and spacious sound. It is often used in music production, especially in genres like ambient, post-rock, and shoegaze, to evoke a sense of vastness and dreaminess.
Key Characteristics of Shimmer Reverb:

Extended Decay: Longer reverb tails that create a sense of space.

Pitch Shifting: The reverb signal is pitch-shifted upward (or sometimes downward) before being mixed back in, adding shimmering overtones.
Harmonic Richness: The pitch-shifting generates harmonics that give the reverb a "shimmering" quality.
Atmospheric Sound: It produces a glowing, almost otherworldly atmosphere.

How It Works:

Typically, a shimmer reverb effect involves splitting the reverb signal into multiple paths & one path remains unchanged (dry signal). The other path is pitch-shifted, often upward by a few semitones. The pitch-shifted signals are then mixed back with the original, creating a shimmering, luminous sound.
Usage:

Musicians and producers use shimmer reverb to add depth and texture to guitars, vocals, and other instruments, especially when aiming for a dreamy or spacious soundscape.
Historical Significance:

The Shimmer Reverb effect, known for its lush, expansive, and spacious sound, has been widely used across various genres by numerous artists. One notable example is the band U2, particularly in their song "Where the Streets Have No Name", where a shimmer reverb effect is used to create a soaring, atmospheric sound.
Another example is Sigur Rós, who frequently incorporate shimmering reverb effects into their ambient and post-rock soundscapes, notably in tracks like "Hoppípolla".
Additionally, the band Explosions in the Sky often utilize shimmer reverb effects to craft their expansive, cinematic sound.

In Summary: A shimmer reverb is a specialized reverb effect that combines long, lush reverberation with pitch-shifted harmonics to produce a shimmering, ethereal sonic atmosphere.

Spring Reverb

A Spring Reverb is an audio effect that creates reverberation (echo-like sound) using a physical spring as the reverb medium. It works by sending an electrical audio signal through a transducer (called a driver) attached to one end of a coil of coiled metal spring.
The mechanical vibrations travel through the spring, reflecting and bouncing along its length, then are picked up by another transducer (called a pickup) at the other end. The resulting signal is a delayed, diffused version of the original sound, producing a characteristic reverb effect.
Spring reverbs are known for their distinctive, metallic, and slightly boomy sound, which has been popular in guitar amplifiers, vintage audio equipment, and certain studio settings. They are valued for their unique tonal qualities and their ability to add depth and space to audio recordings.

Key points:

Uses a physical metal spring to create reverberation. Produces a distinctive, metallic reverb sound.
Commonly used in guitar amplifiers and vintage gear. Provides a characteristic "springy" echo effect.
Historical Significance:

Several music groups and artists have utilized spring reverb effects in their recordings to create distinctive sounds. One notable example is The Shadows, a British
instrumental rock band, who famously used spring reverb to achieve their signature echoing guitar tones, notably on their hit "Apache".
Another prominent example is The Beach Boys, particularly in their early surf rock recordings, where they used spring reverb to create the lush, spacious sound characteristic of tracks like "Surfin' USA" and "Fun, Fun, Fun".
The Ventures also employed spring reverb in their surf and instrumental rock recordings, contributing to the "wet" guitar sound. In the realm of electronic and
experimental music, artists such as Brian Eno and Tangerine Dream have used spring reverb effects in their studio setups to craft atmospheric textures.
Overall, spring reverb has been a popular effect among surf rock bands, instrumental groups, and experimental musicians for its distinctive, resonant echo.

Appendix: Routing

Routing is a key aspect in digital consoles and can be … intimidating, especially with desks such as WING, with a multitude of physical sources and destination, tap points, and total flexibility of signal path arrangements for mixing and routing. "*In the world of digital consoles with a multitude of inputs, outputs, and complex mixing capabilities, routing is the fundamental process that determines how audio signals flow throughout the system. It's akin to a sophisticated traffic control center, ensuring each sound reaches its intended destination with the desired processing applied. Routing in digital consoles is the foundation for achieving a high-quality, well-controlled sound. By mastering this skill, you can unlock the full potential of your console and create professional-sounding mixes with unparalleled flexibility*"[111].

The following chapters along with existing videos on routing you can find on the web will help you with your first steps in routing your signals in the WING console.

WING routing is always done from a WING perspective:

- For input routing, input `SOURCES` are the physical connections to WING, while destinations are either `WING CHANNELS` or `OUTPUTS` (i.e. physical outputs);
- For output routing, signal `SOURCES` are any of the possible tap points in WING to send out a digital audio signal, including `INPUT SOURCES`, `BUS`, `MAIN`, `MATRIX`, `USER SIGNAL`, `MONITOR`, and `FX SENDS`, while destinations are the physical outputs available from the desk or additional devices that are connected to it.

Benefits of Effective Routing:

- **Clean Mixes**: Proper routing avoids unwanted signal bleed and ensures each element sits clearly within the mix.
- **Efficient Workflow**: By creating custom routing setups, you can save time during live performances or studio sessions.
- **Creative Possibilities**: Advanced routing unlocks creative options like sending specific instruments to dedicated effects mixes, or managing both FOH and Monitoring from the same desk.

Understanding Routing Interfaces:
Digital consoles such as WING offer visual interfaces for routing, with screens depicting virtual "patches" connecting inputs, outputs, and internal processing modules. For WING, the interfaces are the main touchscreen using the `ROUTING` dedicated button, or software applications such as `WING-Edit`[112] or `Mixing Station`[113].

[111] Source: Gemini AI
[112] See: https://www.behringer.com/series.html?category=R-BEHRINGER-WINGSERIES, under the Software section
[113] See: https://mixingstation.app/

Input Routing

WING has numerous possibilities when it comes to connecting sources and channels (so called “routing”), effectively enabling audio to ‘flow’ from its source to the WING audio engine for processing and mixing within the desk.

There is a very large choice of no less than 376 input sources that can be found under the ROUTING→SOURCES screen, the SOURCE GROUP selection includes:

- LOCAL IN (8 local XLR inputs on the full-size desk)
- AUX IN (8 local 6.3mm inputs on the full-size desk)
- AES/EBU IN (2 AES/EBU inputs)
- OSCILLATOR (2 internal oscillator sources with various signal options and settings)
- AES50-A, B and C (each with 48 inputs)
- ST CONNECT (StageConnect™, configurable 32 IN or OUT at line level on a standard XLR/DMX cable)
- USB AUDIO (48 inputs from a USB-2.0 port)
- WLIVE PLAY (2x 32 inputs from one or two SD cards)
- DANTE[114] (64 inputs from either a card or internal module, or 128 inputs if both are installed)
- USB PLAYER (4 inputs from the USB stick input)
- USER SIGNAL (48 configurable user data path or patches overlapping/referencing sources above)

All Sources above come with their associated SETTINGS *(Mono/Stereo/MS, Gain, Polarity, Mute, Phantom, …)*, ICON, COLOR, and TAGS.

The process of “source routing” or “input routing” is the action of associating a SOURCE to one of the WING 48 (input/aux) Channels (or Channel Strips) for mixing their audio within the desk. This is accomplished by pressing the ROUTING button on the left of the WING Screen, and selecting the CHANNELS tab on the screen, opening the following screen:

There are two routing options: MAIN and ALT. Both serve the same purpose and can be selected within Channel Strips; There are therefore two routing tables available at the desk.
WING routing tables are write-protected (to avoid major issues during live performances) unless the unlocked

[114] This could be another option

padlock [🔓] is selected. With the padlock being green, a Channel Strip can be selected in the left side of the screen and a SOURCE entry can be selected from the Sources available on the right side of the screen.

Different groups of SOURCES blocks are available from the SOURCE GROUP pull down menu (see below):

SOURCES include actual HW sources and logical audio paths such as BUS, MAIN, MATRIX and USER SIGNAL, which are either the result of partially mixed audio or a specific/customized selection of Source in the case of USER SIGNAL.

After a console init as shown above, LOCAL IN 1…8 mono sources will already be routed to Channels 1…8, USB AUDIO 1&2 will be combined as a stereo source routed to Aux 1, and Aux 2 will receive USB PLAYER 1&2 as a stereo source. This constitutes the default MAIN routing table. The ALT routing table is empty.

For example, routing WLIVE PLAY sources 1..16 to Channels 9..24 can be done by a click on the +1AUTO button, selecting the first Channel to modify routing for (9), selecting WLIVE PLAY in the SOURCE GROUP pull-down menu and sequentially clicking on the 16 first entries of WLIVE PLAY, resulting in the following screen (and MAIN routing table).

Additional, similar, or different choices for routing could be done for the ALT routing table, offering an alternate set of SOURCES to mix from at the mixing desk. Note that the MAIN/ALT selection at the Channel Strip level is accomplished by selecting either the MAIN or the ALT source for that Channel, i.e. moving from one to the other will possibly select a different source to mix, losing the previous one.

At that point, Channel Strips 9 to 24 can be used to mix the audio data issuing from the first 16 tracks of SD card 1[115]. Channel Strip 1 to 8 can be used to mix the audio data coming from local inputs 1 to 8 at the back of the console, and Aux strips 1 and 2 can be used to mix the audio signals from USB 1 & 2 and the 2 tracks from the USB stick player.

What if the 16 `WLIVE PLAY` tracks above were representing 8 distinct stereo channels?

`WLIVE PLAY` sources must be declared as stereo pairs; This is done by returning to the `SOURCES` tab and selecting the `WLIVE PLAY` source group, showing all 64 possible entries. Selecting entry `1` and clicking on `STEREO` in the `SETTINGS` will automatically 'join' entries `1` and `2` as a stereo pair named `CRD1/L` and `CRD2/R`, the same action can then be done for entries `3, 5, 7, 9, 11, 13`, and `15`, resulting in 8 pairs of stereo sources that can be routed to 8 different channels as described earlier. The pictures below show the screens resulting from the actions we just described.

[115] Note that SD card 1 maps to entries 1..32, and SD card 2 maps to entries 33..64 in the `WLIVE PLAY` or `REC` screens

As a result from the operations above, Channel Strips 9 to 16 can now be used to mix the audio data issued from the first 16 tracks of SD card 1. Channel Strip 1 to 8 can still be used to mix the audio data coming from local inputs 1 to 8 at the back of the console, and Aux strips 1 and 2 can still be used to mix the audio signals from USB 1 & 2 and the 2 tracks from the USB stick player.

Output Routing

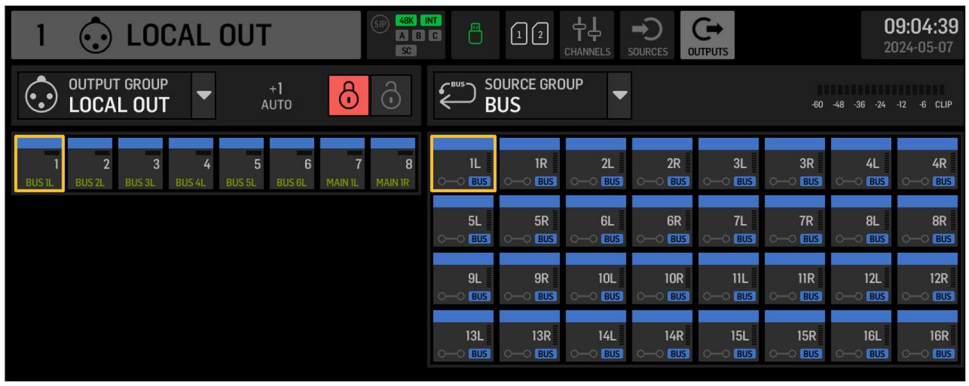
Output routing works in a similar way to Input routing. This time though, the `OUTPUTS` tab is selected in the `ROUTING` screen, revealing `OUTPUT GROUPS` from a pull-down menu, and representing the physical outputs where audio signals from the console can be routed to, using digital audio sources selected from the `SOURCE GROUPS` pull-down menu and entries.

The selection of output physical connections is as numerous as for inputs and characterizes the console versatility and extended capabilities with 374 physical outputs that can be found under the `ROUTING→OUTPUT` screen; The `OUTPUT GROUP` selection includes:

- `LOCAL OUT` (8 local XLR outputs on the full-size desk)
- `AUX OUT` (8 local 6.3mm outputs on the full-size desk)
- `AES/EBU OUT` (2 AES/EBU outputs)
- `AES50-A, B and C` (each with 48 outputs)
- `ST CONNECT` (StageConnect™, configurable 32 IN or OUT at line level on a standard XLR/DMX cable)
- `USB AUDIO` (48 outputs from a USB-2.0 port)
- `WLIVE REC` (2x 32 outputs to one or two SD cards)
- `DANTE`[116] (64 outputs from either a card or internal module, or 128 outputs if both are installed)
- `RECORDER` (4 outputs to the USB stick input)

As for source assignments, the console comes with a default output routing right after being initialized. This is shown below with `LOCAL OUT 1..8` receiving `BUS 1L` to `BUS 6L`, `MAIN 1L` and `MAIN 1R`, respectively. Note also that `AUX OUT 7&8` are receiving `MONITOR A&B` by default.

[116] Could be another option



Changing the output routing is made from the `ROUTING→OUTPUTS` screen, with first clicking on the `OUTPUT GROUP` of interest to select the first physical destination you want to assign a WING signal to.

Say we would like to record the first 8 (all stereo) channels of our show as a new session onto SD card 2, along with the resulting show mix on the USB stick of the front panel as a 2-track mix coming from MAIN1; The operations one would perform are as follows:

After selecting the `WLIVE REC` group of physical output on the left side of the screen and clicking on any entry in that panel, we need to select where routed signals will be coming from. That selection is possible from the `SOURCE GROUP` pull-down menu on the right side of the screen that offers a list of all possible 'tap points' for getting digital signals from the desk to physical outputs. In our example, we would select the audio signal sources that feed our Channel Strips 1 to 8. Let's further assume our 8 Channel Strips are taking their inputs from `WLIVE PLAY` as described above. SD card 2 maps its 32 entries from `33` to `64` in the `WLIVE PLAY` or `REC` panels. As we did for input routing, we first select the unlocked padlock [], click on the first entry for SD card 2 (`CRD 33`) in the `WLIVE REC` panel on the left side of the screen, click on the `+1AUTO` button for ease of selection and choose our audio signals to route to our outputs by clicking on the 16 entries in the `WLIVE PLAY` panel on the right side of the screen, starting at entry `CRD1`, resulting in the following routing table:

The audio data used for our mix will be recorded to SD card 2, from SD card 1 (only when engaging record on SD card 2 and play on SD card 1, of course).

We also need to set the output routing for recording our live mix; In the `ROUTING→OUTPUTS` screen, with clicking on the `OUTPUT GROUP` on the left side of the screen, we select `RECORDER` and click on the first entry, `1`. With the unlocked padlock [🔓] selected and the `+1AUTO` button engaged, we select `MAIN` in the `SOURCE GROUP` on the right side of the screen, and then click on the `1L` and `1R` entries in the displayed table. This completes our output routing with the following screen:

We can now select the USB recorder and start a 2-channel recording on USB stick, then select the SD card screen, start recording on card 2, select our 8 stereo tracks session and hit play on card 1 and mix our session, simultaneously getting a digital copy of our dry data from SD1 to SD2 and a live mix result as a stereo wav file in the USB stick.

Advanced Routing Options

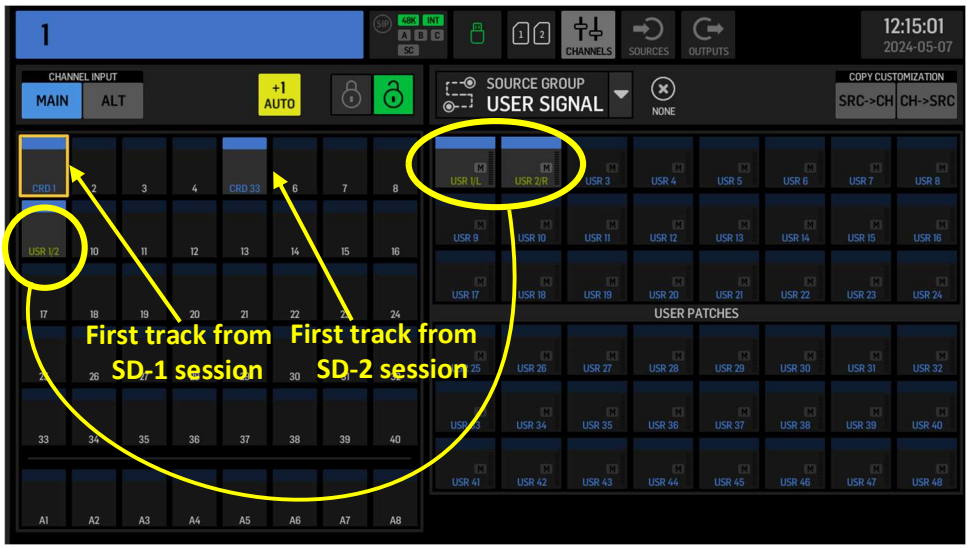
All routing scenarios presented above have a restriction when it comes to stereo pairs. The HW limitations of the desk impose stereo pairs to always be in the form [odd-even] SOURCE numbers; i.e. you cannot route a stereo signal to a single channel strip if your two mono sources are connected to say LOCAL IN 2 and LOCAL IN 3, or if they are connected to WLIVE PLAY 1 and WLIVE PLAY 33.

This is where USER SIGNALS come into play, leveraging the internal WING FPGA routing chip flexibility to remove some of this restriction. A USER SIGNAL is a virtual channel, and proposes two variants: USER SIGNAL and USER PATCHES which we'll detail below:

USER SIGNAL

A USER SIGNAL can only accept INPUT, AUX or BUS, MAIN or MATRIX channels as source. Setting or assigning sources to USER SIGNAL is done with selecting the ROUTING->SOURCES screen and choosing one of the 24 USER SIGNAL entries in the SOURCE GROUP pull-down menu on the left side of the screen.

Clicking on an entry will display the SOURCE characteristics on the right side of the screen, with SOURCE SETTINGS, ICON, COLOR, POLARITY, and MUTE, and a +ASSIGN button that is used for selecting which channel, tap point (TAP or POST) and whether using stereo, mono, or M/S data as signal(s) for the selected USER SIGNAL entry. If the selected USER SIGNAL entry is stereo, it is possible to choose two totally disjoint sources for each of the L and R paths of the selected USER SIGNAL, such as for example channel 1 and channel 5 that would be routed with WLIVE PLAY 1 and WLIVE PLAY 33 to take our example above, creating a stereo pair that can now be assigned/routed to a single Channel strip thanks to USER SIGNALS. The screenshots below show routing displays for such a case, with channels strips 1 and 5 routed to sources WLIVE PLAY 1 & 33, and channel strip 9 routed to stereo USER SIGNAL 1 that routes channels 1 and 5 as a single stereo pair to itself.



That same `USER SIGNAL 1` can also be used as a `SOURCE` for an `OUTPUT`, such as for example to record into a DAW on a PC using a USB connection, as a single stereo pair into `USB 1&2`.

To achieve this, we would click on the first entry of `USB AUDIO` in the `OUTPUT GROUP` on the left side of the `ROUTING→OUTPUTS` screen. With the unlocked padlock [🔓] selected and the `+1AUTO` button engaged, we select `USER SIGNAL` as a `SOURCE GROUP` on the right side of the screen and click on the `USER 1L` and `USER 2R` entries in the displayed table. This completes our output routing with the following screen:

We can then hit play on WLIVE sessions on both SD card 1 and 2, and will get our signals available as a single stereo pair for recording a 2-track session over USB cable to a connected PC.

USER PATCH

A USER PATCH can be used the same way as a USER SIGNAL but unlike USER SIGNAL, the audio physical SOURCE will directly connect to a USER PATCH, thus bypassing the need for using an intermediate channel strip or set of channel strips.

Setting or assigning sources to USER PATCH is done with selecting the ROUTING->SOURCES screen and choosing USER SIGNAL in the SOURCE GROUP pull-down menu on the left side of the screen and selecting one of the 32 USER PATCH entries in the list. As for USER SIGNAL, clicking on a USER PATCH entry will display the SOURCE characteristics on the right side of the screen, with SOURCE SETTINGS, ICON, COLOR, POLARITY, and MUTE, and a +ASSIGN button that is used for selecting which physical SOURCE will be used. A USER PATCH can accept any of LOCAL IN, AUX IN, AES50 A/B/C, ST CONNECT, USB AUDIO, Add-on Card, Internal Module, USB PLAYER or AES/EBU IN signal as its routed SOURCE.

If the selected USER PATCH entry is stereo, it is possible to choose two totally disjoint sources for each of the L and R paths of the selected USER PATCH, such as routing for example WLIVE PLAY 1 and WLIVE PLAY 33 to USER PATCH stereo entry 26L/26R to use once more our example above, creating a stereo pair that can now be assigned/routed to a single Channel strip.

The screenshots below show routing displays for such a case, with sources WLIVE PLAY 1 & 33 (our installed Add-on Card) routed to USER SIGNAL 25L/26R, itself routed as a single stereo pair into channel 9.

USER PATCH has the advantage of routing simplicity over USER SIGNAL. On the other hand, USER SIGNAL offers more signal processing or mixing capabilities over USER PATCH, to the expense of using intermediate Channels.

Appendix: Shows, Scenes (Snaps, Snippets, Presets & Audio Clips)

The WING desk has a high level of functionality to manage saving and restoring `Shows`, `Snaps`, `Snippets` and `Presets`, or `Scenes`.

Shows

A key feature in digital consoles is their ability to save and restore state (in different forms) to easily change from one set to another, or save work for later use. This helps maximize the use of the desk in situations where several bands share the same console, or in recording studios where saving console state is a must have for effectively managing recordings and customer data.

For WING, a Show is typically a collection of up to 1000 Scenes. `Show` files contain references to `Scene` entities, and not a copy of the actual data.

`Shows` can be managed directly from the WING screen via the `LIBRARY` button or using `MIDI` commands (see below). One can create a `Show`, open, or delete it. There can be only a single active `Show` at any time.
`Snaps`, `Snippets`, `FX` or `Channel Presets` or `Audio Clips` can be added to the current `Show`, they can also be re-organized using the options provided in the `LIBRARY` section. When added to a `Show` file, they are referenced as `Scenes`.

Users can 'navigate' up and down [i.e. loading `Scenes`] in the current `Show` using the `Show Control buttons` dedicated to that effect [`GO`, `NEXT`, `PREV`, `GONEXT`[117], `GOPREV`[118]], `OSC` or `MIDI` commands, or `wapi` calls. Some of the `Show` items can also be marked as '`skip`' for a quick avoiding loading them during navigation. `Show` items can also be marked with a '`link`' tag to enable simultaneous loading of multiple items during navigation.

Please refer to the Behringer documents on how to use `Shows`[119]. When loading a `Scene` by mistake, the `RESTORE` button can be used to return to previous console state.

Scenes

A `Scene` can represent anything used in a `Show`. It can refer to a `Snap`, a `Snippet`, a `Channel` or `FX Preset` or an `Audio Clip`, or a combination thereof. Each single entry in a `Show` file is a separate `Scene` that can be loaded using the `LIBRARY` navigation options.
`Scene` names can be built in two parts, separated with a '~' character (like in "`SnapName~Explanation`" and be displayed on two lines, with the part left of the '~' char being in large font letters and the part right of the '~' char displayed as a second line in smaller characters. This can be handy for adding explanation or use for the Scene element to name (see below).

[117] GONEXT means Go to Next item, i.e. the next item is first pointed to and a GO command is then executed.
[118] GOPREV means Go to Previous item, i.e. the previous item is first pointed to and a GO command is then executed
[119] Available at: https://mediadl.musictribe.com/download/software/behringer/WING/WING_Firmware_1.13_GUI-Description.pdf

Snaps (& Scopes)

A Snap file contains the full set of WING parameters, optionally associated with Scopes that list the set of parameters of interest at the time the Snap was created, thus limiting the effect of loading a Snap file to a subset of parameters selected with the Scope.
Scopes can be changed at load time if needed. They can also be modified as needed and saved. Scopes will apply at the time the Snap file is loaded.
Snap files are very important in for WING users as they are the simplest way to save their work (i.e. the full state of the console) under a single file.

There may be differences though if saving a Snap file on a WING and then loading it on a different type of WING (Standard vs. Compact vs. Rack); Main differences are with the use of USER Layers and CC definitions where there are obvious differences between the 3 WING console types. One has to assume USER layers and CC definitions are save for a particular type of WING and will not show up on a different one;
Note that they will be ignored, not erased or overwritten; So for example, a Snap file saved with Standard WING layers can be loaded onto a WING Rack where new Layers definitions can be added and saved again as a new Snap file which then will be valid on both types of consoles.

Snippets

A Snippet file allows recording of any WING parameter changes as well as manually adding/removing of parameters using the LIBRARY buttons ADD ITEMS and REMOVE ITEMS.

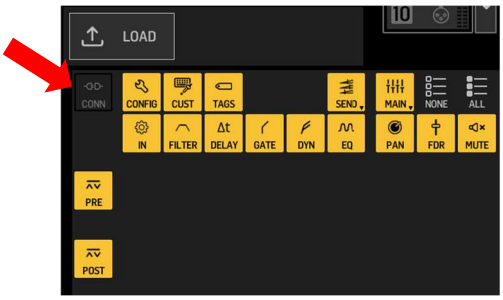
- REC FOCUS defines which parameters are observed during REC active.
- LOAD FOCUS allows loading a parameter set from any existing snippet.
- When a snippet is saved or updated, the *current* values of all parameters (in FOCUS) are written to the file and cannot be changed once recorded to INT or USB file.

Presets

A Preset file allows recording of WING parameters specifically targeting FX or Channel attributes.

- FX and CHANNEL Presets
    - Presets contain target FX slot(s) and target channel/scope information; When used within a Show, the Preset data is instantiated within the Show as a Scene, and the same Preset can be used to load different FX engines / channels (with different scope). After adding a Preset to a Show, just change settings and click UPDATE SCENE (don't forget to save the show).
    - If you set the target FX slot of a Channel Preset (one of the inserts) to NONE, the insert is switched off when loading the Preset.
    - Premium effects can only be loaded into FX engines 1-8.

- CHANNEL Presets
    - Gain and Phantom power status which are part of the source associated to the channel used to create the preset are saved with the Presets, but are not loaded by default when applying the preset to another channel; You will have to enable/select the "conn" setting box (see red arrow below) to ensure Gain and Phantom power are restored; This will also affect the source to the destination channel.



- If you set the target `FX` slot of a `Channel Preset` (one of the inserts) to `NONE`, the insert is switched off when loading the `Preset`.
- `Presets` can contain insert effect data; Care must be taken when loading them, as effect engines might be used in other `Channels`.
- `Channel/Aux/Bus/Main/Matrix Presets` can only be loaded into corresponding `Channel` of course.
- `Bus Presets` contain `Channel` feeds into the bus (`FEED` scope).

- Old `ROUTING` Presets can be loaded as `Snapshots` (scope is set accordingly). For new routing `Presets`, just use `Snapshots` and use `Scope` to limit loading to routing parameters only.

Audio Clips

WING `Show` control enables using Audio Clips as `Scene` entities. One can therefore include a reference to a .wav file from a USB stick or stored in WING's internal file system as a Scene that can be part of a Show file. Library navigation functions can be used to launch (load) Audio Clips that are part of a Show as they do for any other Scene entity.

Controlling Scenes and Shows via CC buttons

When part of a `Show`, `Scenes` can be loaded using `CC` buttons, rather than using the console `Show` control commands [`GO`, `NEXT`, `PREV`, `GONEXT`, `GOPREV`]. To achieve this, one must first assign a `tag` to `Scene` elements that will be controlled using a `CC` button, using the `EDIT TAG` button under the `SHOW` screen, and assign a `tag` beginning with "`#`" and followed with a number. When going to the `CC` controls, navigating to function `SCENE RECALL`, it is possible to select a `SCENE TAG` corresponding to the tag assigned to the `Scene` to load.
Be aware loading a `Scene` can overwrite `CC` buttons, and therefore you must protect them from being overwritten, either by unselecting `CC` in the `Scene Scope` (if applicable) or with using `Global Safes` to ensure the `CC` area is left untouched when loading `Show` items.
Also, jumping to a `Scene` using a `CC` will affect the order of your items a `Show` currently points to as you will effectively 'jump' to that `Scene` in the `Show`. I.e. if you for example have `Scenes 1`, `2`, `3` and `4` in a `Show` and are currently at `Scene 1`, using the `GO` button to move from `Scene` to `Scene`. Using a `CC` to load `Scene 3` will have the same effect as skipping `Scene 2` and directly go to `Scene 3`, and your current `Show` element will be `Scene 3` with a `GO` command moving to `Scene 4`.

Controlling Scenes and Shows via MIDI

As mentioned in the MIDI chapter earlier in this document, Scenes and Show control can be managed using MIDI commands sent to MIDI channels 7, 8 and 9 as below:

MIDI Scene Change (on MIDI Ch 7):

CH7 CC0 (bank MSB), CH7 PC 1..128 → Scene number 1..128 on bank MSB 0, number 129..256 on bank MSB 1, etc.	B60000..B60008, C600..C67F

MIDI Show Control (on MIDI Ch 8 & Ch 9):

CH8 CC0 (bank MSB), CH8 PC 1..128 → Scene tag #1..#128 on bank MSB 0, tag #129..#256 on bank MSB 1, etc.	B70000..B7007F, C700..C77F
CH9 PC 1→ Scene GO	C800
CH9 PC 2→ Scene PREV	C801
CH9 PC 3→ Scene NEXT	C802
CH9 PC 4→ Scene GO PREV	C803
CH9 PC 5→ Scene GO NEXT	C804

LIBRARY items/scenes can be recalled *by their number* with MIDI Patch Change commands (including Bank MSB when > 128) on MIDI Ch7 with SETUP→MIDI REMOTE CONTROL→SCENE CHANGE enabled. As a result, one can address 128 Scenes by their number using Patch Change on MIDI Ch7, and all 1000 Scene numbers above 129 via Bank/Patch Change on MIDI Ch7.

> The use of Ch7 Bank MSB (to select scene numbers > 128) is only valid when more than 128 scenes are present/included in an active show. When less than 128 scenes are present, any combination/value of ch7 Bank MSB will revert to selecting scene numbers 1..128.

Therefore, and unless using a pure sequential recall of scenes with GONEXT/GOPREV in a Show, scene tags can provide a better option for ensuring the right scene/item is selected/recalled.

Item Tags

Scenes can be 'tagged', providing alternative MIDI or CC button recall options. Tags work as follows:

- A tag can be added to any Library item (Scene, Snip, Clip, Presets or Audio Clip) from the LIBRARY→SWOW screen.
- LIBRARY items/scenes can be recalled *by their tag* #1 .. #128 [to match with MIDI data 0…0x7f] with MIDI patch change commands on MIDI Ch8 with SETUP→MIDI REMOTE CONTROL→SHOW CONTROL enabled, or with custom control buttons [using the SCENE RECALL setting for said buttons]. You can recall tags #1..#16384 using the combination BankChange[MSB only],PatchChange; For example, B70000C702 is targeting scene tag #3, B70001C700 is targeting scene tag #129, etc.
- Scene tags are not necessarily in the same order scene numbers and offer an alternate and more secure method for recalling library items.

- A same tag can be assigned to several `Library` items; In that case, WING doesn't check for exclusiveness of tags and the first one found in the list of `Library` items wins.

Arbitrary MIDI data

Additionally, you can add/send arbitrary `MIDI` data with each `Scene` recall (use hexadecimal notation, separator is optional, i.e. `C002` or `B0,01,7F`). This arbitrary `MIDI` data can be saved with an empty `Snippet`, enabling a very flexible control of external `MIDI` devices directly from the console. This is achieved with using the `SEND MIDI` button under the `LIBRARY→SHOW` screen, and enter arbitrary `MIDI` data in the `EDIT MIDI TEXT STRING` window that opens on the console screen.

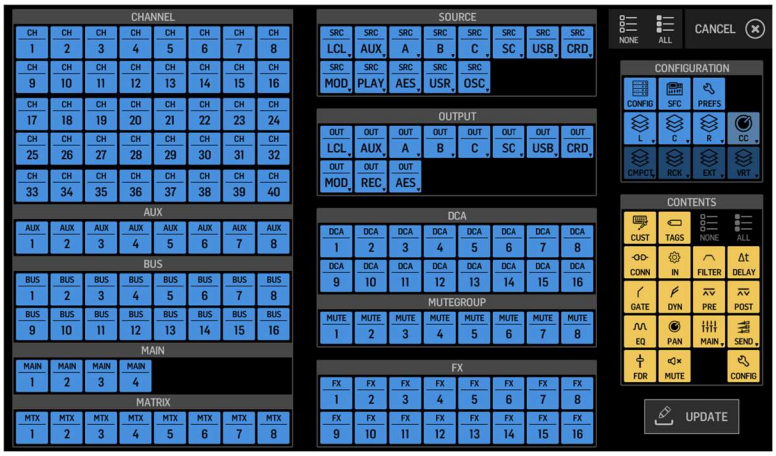
Appendix: Scopes and Safes

Scopes are specific indicators that are used to focus (or restrict) an operation on certain parameter sets of the console when dealing with Scenes or at `INITIALIZE CONSOLE` time.

Safes are specific indicators that are used to prevent the modification of selected parameter sets of the console when dealing with Scenes

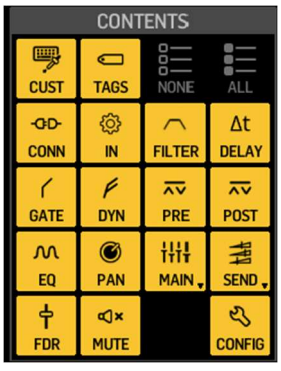
Library Scopes[120]

When editing scopes during a Library action, a list of icons displays on the screen as shown below:



Most of them are explicit, but some (listed below) regroup several items or parameters under a single icon that can be selected or un-selected depending on the scope edit or recall operation the user wants to perform. The paragraphs below list the different parameters that are covered by these icons.

CONTENTS Scopes (orange Icons)

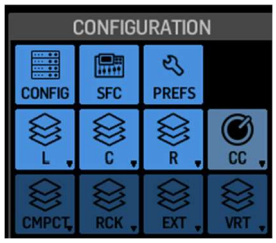


CUST: Icon / Name / Color / Light on, off
TAGS: Custom Tags / DCA, Mute, Talk Tags
CONN: Source A, B / Input Select Status
(No source Mute or Mono-Stereo-MS)
IN: Main, Alt Status / Trim / Balance / Phase flip
FILTER: LPF / HPF / TILT (Max, AP90, AP180) with all settings
DELAY: On, Off Status / Delay Time
GATE: All settings of the gate (Type / Settings / Side chain …)
DYN: All settings of the dynamic (Type / Settings / Sidechain …)
PRE: Assignment of the FX Plugin (without FX settings)
POST: Assignment of the FX Plugin (without settings), Automix Settings (X, Y, Amount)
EQ: All settings of the EQ (including type of EQ bands), TAP EQ Settings in Bus Sends
PAN: Pan and Width Settings
MAIN[1..4]: Levels/ On, Off Settings / Pre, Post Settings

[120] Many thanks to *Andy Lauer* for providing these details.

SEND[Bus 1..16, Mtx 1..8]: Levels / Pan Settings / All status (On, Off / Mode Link / Send Pan / Send Mute / Mode)
FDR: Fader Levels
MUTE: Channel Mutes (no Source Mute)
CONFIG: Process Order / Tap Point / Solo Bus Status

CONFIGURATION Scopes (blue icons)



CONFIG

Monitor Page - Monitor Control: Mute Status, Output Status
Monitor Page - Talkback: – Talk Channel Assign, All Talkback Preferences
Monitor Page – Monitor A and B: All settings and Align options (Delay / EQ / Invert)
Setup – Audio: Main Link, DCA Mutegroups, Startup Main Mute, Automix X - Y (enable status), Solo Mode, Channel Solo, Bus Solo, Main Solo, Matrix Solo, Source Solo
Setup – Surface: Main Meter Dropdown, Main Meter Tap, Show Source On Scribble
SD Card: All settings except "Link Status"
RTA: Range, Decay, Detector, Autogain on/off, Fixed gain value

SFC

Setup – Audio: Mutegroup/SIP Override, Exclusive Solo, Solo Follows Select, Select Follows Solo, BUS/MAIN SoF Activates Solo
Setup – Surface: All The "Lights" settings, Full Fader Paging, Channel Meters, Bus Meters, Main Meters, Matrix Meters, DCA Meters, Screen Follows Ch Strip, Ch Autoselect, User Layer Link, Use F1- F3 As Custom Controls, Right Section Sends On Fader, SOF Button, Show SOF Frame, Alternative SOF Mode, Sel Dbl Click

PREFS

Setup – General: Show Meter Page When Locked, Use CRSR/WHEEL For Parameters, Touch Fader Select, Touch Fader Res, Mouse Disables Touch, Mouse Speed value
Setup – Surface: Tap Tempo Flash, Fader Speed
Setup – Remote: Complete Midi Remote Control
Setup – DAW: All DAW Settings

L, C, R, CC, CMPCT, RCK, EXT, VRT

Refer to layers in the console. Some are specific to the Compact and Rack models.
CC refers to the Custom Controls that can be edited in your console.

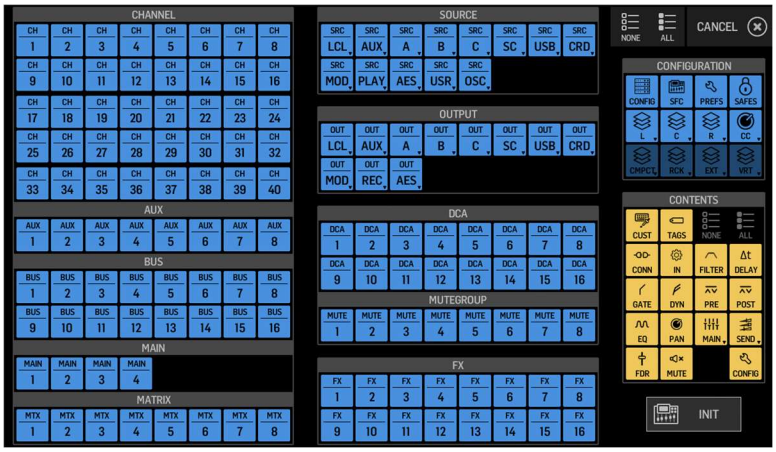
Not Saved in Snapshots:

Monitor Page – Monitor Control: DIM and MONO Button, TALK A and TALK B Button
Setup – General: Console Name, Time Date, USB Host Speed, Confirm Library Load, Confirm Library Update
Setup – Audio: Audio Clock (Rate and Sync Source dropdowns), "INPUT SELECT" switch status, Startup Main Mute, Global Input Select Override, USB AUDIO (In/Out dropdown)

Setup-Remote: HA Remote (All settings), Network Settings (dropdown incl. addresses), Remote Lock (OSC / TCP)
4 Track Recorder: Settings (2/4 Ch – 16/24 Bit)
SD-Card: Link Status

Console Init Scopes[121]

In the INIT screen, a screen (like the Library Scopes one) will display the following, along with a large INIT button.



the following settings/parameters will only be initialized/"recalled" if you initialize the desk with **ALL** parameters/settings selected. If anything is taken out of the initialization scope the settings below won't be initialized.

- Clock rate and (sync) source
- Global input select
- USB Audio channel configuration
- Startup main mute
- Global input select override
- HA Remote settings
- The OSC Setting in Setup -> Remote -> Remote Lock
- The DAW control preset.

The following settings will never be initialized/"recalled".

- Console Name
- USB Host Speed
- Clock
- Everything in Setup -> Remote -> Network (IP Adress etc.)
- The TCP Setting in Setup -> Remote -> Remote Lock
- Talk, Headphone and Monitor level (physical knobs on the surface)
- Monitor Mono and Dim (physical buttons on the surface)
- Talk A and Talk B on/off (physical buttons on the surface)
- Everything in the Library

[121] From @sinste on the https://behringer.world forum

Global Safes

At the top right of the “LIBRARY” screen, is a sign that can take one of the two following icons/colors:

, or 

, depending on the contents of the referring page. This is used for **Global Safes**.

Global Safes are a series of *parameter indicators* that are used to prevent the modification of the values of the respective console parameters. They are listed as a screen of icons as shown on the right:

They are grouped under classes such as “CHANNEL”, “AUX”, “BUS”, “MAIN”, “MATRIX”, “SOURCE”, “OUTPUT”, “DCA”, “MUTEGROUP”, “FX”, and “CONFIGURATION”

When clicking on a *parameter indicator*, it will turn red and change the state/color of the Global Safes logo on top of the screen, reminding you that *at least one* Global Safe is engaged.

Each parameter indicator represents all the parameters belonging to a CHANNEL, an AUX, ..., or an FX. Some indicators will represent a set of configuration parameters, such as CONFIG, SURFACE, ..., CC [Custom Controls].

When selected [RED], they will **prevent the update/modification** of their respective section when executing a scene **LOAD** operation (snap or snip), or one of the **GO** functions. As a result, Global Safes are a great way to protect certain sections of your console while running a show and using the Show functions of the console.

Note that the **RESTORE** (in LIBRARY) and **INITIALIZE CONSOLE** (in SETUP) functions do not take Global Safes into account and will modify/re-initialize them.

Appendix: WING Startup Control

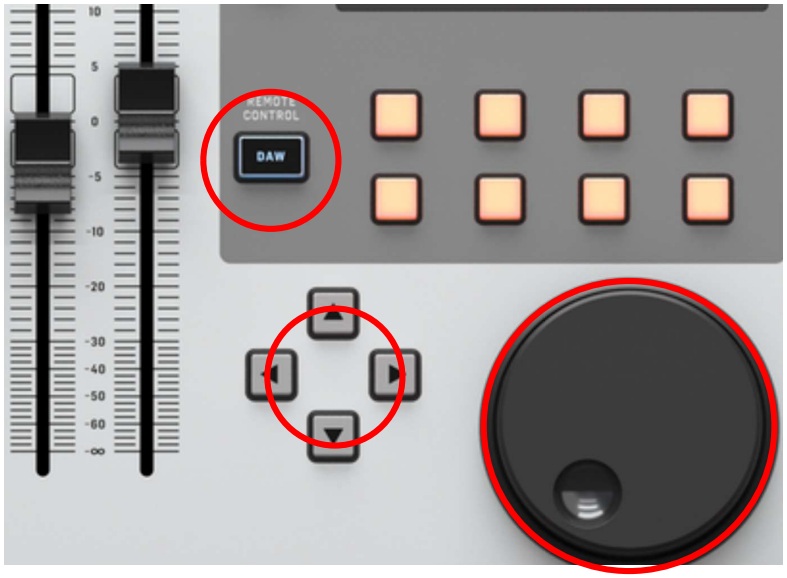
During startup, the console will automatically load a `Show`, `Snapshot` or `Snippet` with the following name when placed in a folder called `STARTUP` in the root of the internal data partition. This can be bypassed when holding the `LIBRARY` button during power up. Files have to start with the letters "`STARTUP`", such as

- `STARTUP.show`, or `STARTUP_myfile.show` for ex.
- `STARTUP.snap`, or `STARTUP_myfile.snap` for ex.
- `STARTUP.snip`, or `STARTUP_myfile.snip` for ex.

When such files exist, you can bypass them at startup time by holding the LIBRARY button while powering the console up.

Appendix: MIDI DAW mode for REAPER Control Surface Use

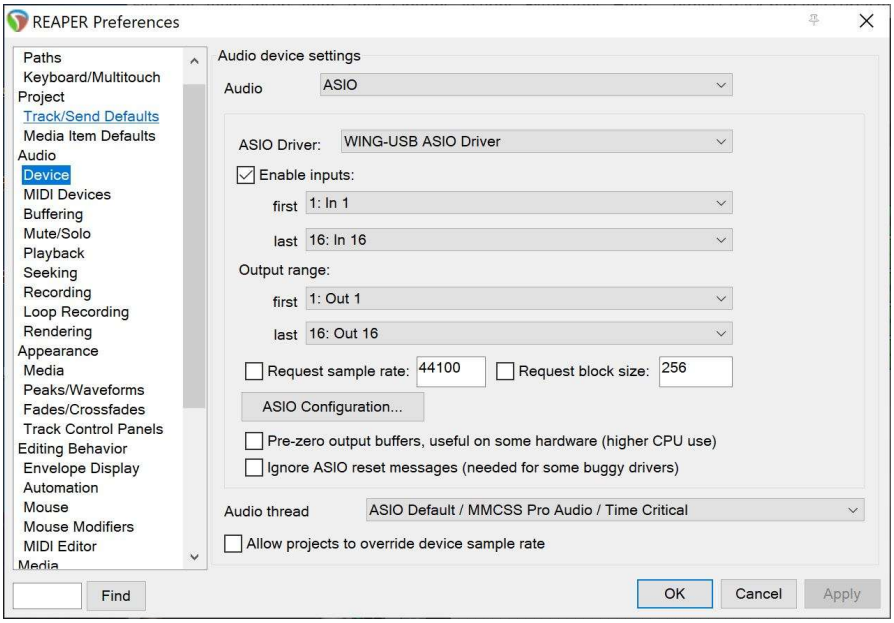
This section is not directly related to programming, but can prove useful when it comes to using WING in a studio, with REAPER™ as a companion `DAW` software.
The simplest and most complete way to connect all elements together is to use MIDI over USB, MCU mode. This will not only provide a link for REAPER's audio to be sent to WING for audio processing, but will also enable several MIDI channels that can be called for using WING as a control surface and transport controls for REAPER.
To achieve this, you will first make sure you have a USB connection between your WING and PC.
You can at any time flip between WING controls and MIDI DAW control using the `DAW Remote Control` button circled in red below and situated left of the group of 8 buttons above the Jog Wheel:



Shown above: the `DAW Remote Control` button, the `Jog wheel`, and the `4 directional keys` mentioned in the coming pages.

REAPER Audio Setup

You then adjust REAPER ASIO interface (and WING setup) to get ASIO channels for audio. The routing on your WING must map USB Inputs to your channel strips. Faders for channel strips should be ideally set to `0dB`. Main strip fader should for the time being be set to `-oo`.
The figure below shows an example for a 16 in/16 out ASIO setup `(Options→Preferences→Audio→Device)`.

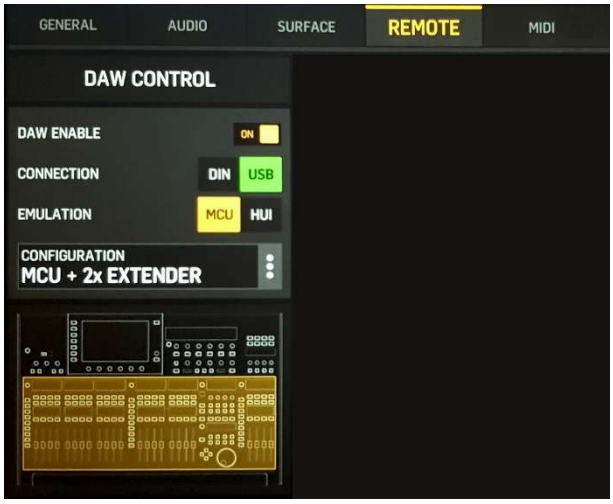


MIDI

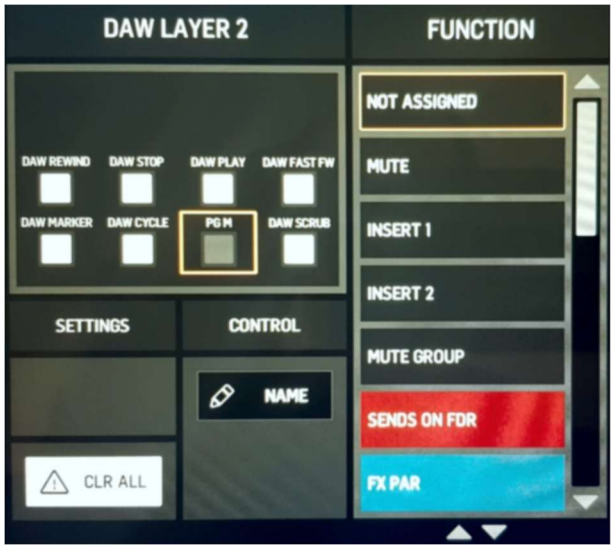
MIDI includes two parts (besides the USB connection mentioned above). The first one is relative to setting WING as a DAW control surface, the second one relates to transport controls.
REAPER DAW control surface is obtained through the `SETUP→Remote` screen. In the left part of the screen, you will choose `USB MIDI` and `MCU+2xExtenders` for a full 24 strips DAW control.

WING MIDI setup

See below the corresponding WING setup screen which can be set from the `SETUP->REMOTE` WING screen. We show here the setup for using a full 24 WING channel strips for MIDI remote control of REAPER, using the MCU + 2 distinct extenders over USB MIDI. You can limit the surface to the controller or controller + 1 extender.



Transport controls proposed here include REW, Fast Forward, Stop/Play/Pause, Scrub, Jog Wheel and more, directly from the lower section of the WING controls. A simple/classic example of implementation is shown below and the setup of these functions is performed after pressing the WING controls' `View` button and assigning keys one by one using the WING main LCD screen.

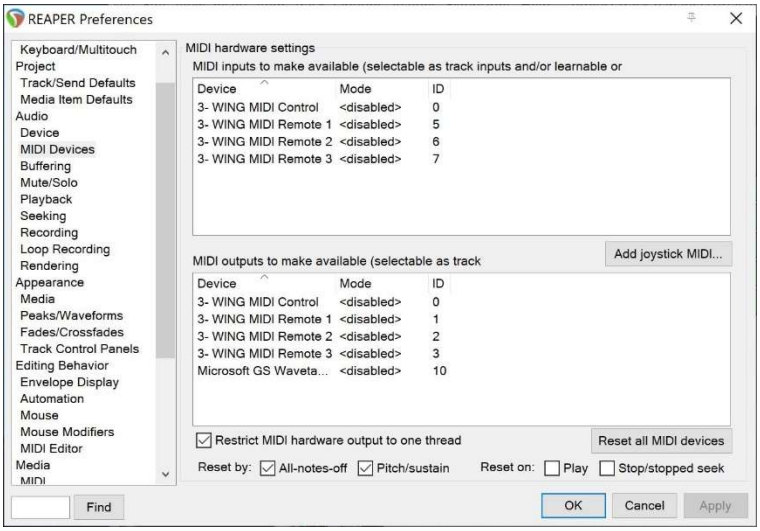


Once modified, the custom transport button layouts can be saved as WING presets. A simple REAPER Control Surface preset is available for download at https://drive.google.com/file/d/1WpAKkxgASSe-X6bIlRm5-7QDyxriDRuR/view?usp=drive_link, resulting in the following "DAW LAYER 2" Control Section assignments[122]:

[122] The 3 other layers, and the rest of the console settings, are left untouched

REAPER MIDI setup

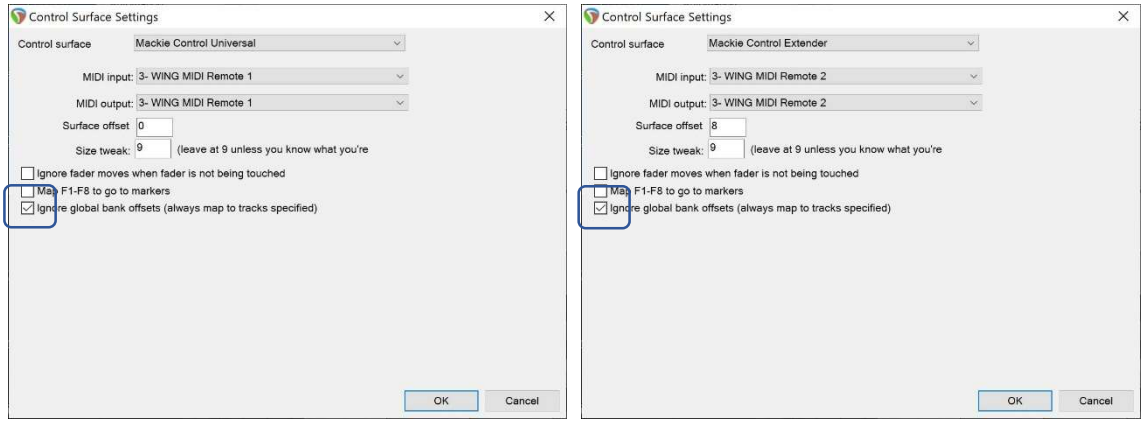
REAPER needs a few simple MIDI settings to correctly enable WING acting as DAW control surface.
When USB is connected, 4 WING MIDI devices appear in the REAPER MIDI devices panel, accessible under `Options→Preferences→Audio→MIDI Devices`. The `MIDI ID` values are managed by REAPER, but can be set as needed, making sure active/enabled device numbers don't duplicate from one active MIDI device to another. This is shown below:



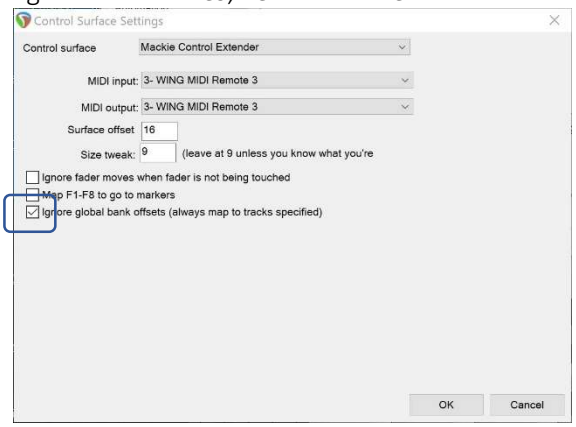
In the case of DAW control use, all WING MIDI devices above must remain `<disabled>` in the REAPER MIDI Devices panel to be used as a control surface communication MIDI device; REAPER will report errors otherwise.

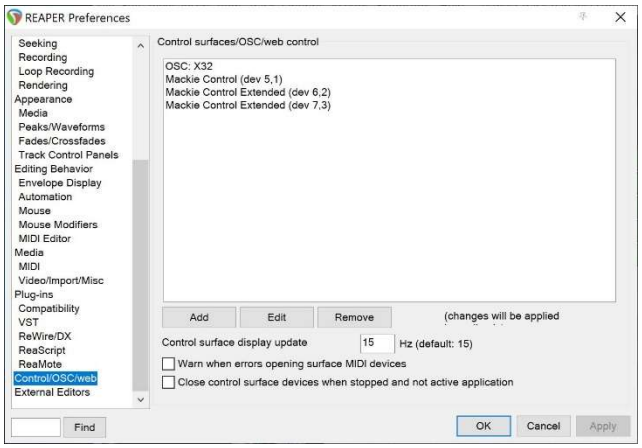
Remember we have setup WING as `USB/MIDI`, `MCU+2xExtenders` to cover three times 8 faders, so the full set of channel strips of WING can be used as surface control strips for REAPER.
In the REAPER control Surface panel `(Options→Preferences→Conrol/OSC/Web)`, you will need to add three separate MCU controllers, the first one is a `Mackie Control Universal` device. Controllers 2 and 3 are `Mackie Control Extender` devices. Each device will connect to a `WING MIDI remote device [1, 2, 3]` respectively, ensuring the `surface offset` parameter is set accordingly to its respective WING group of 8 channel strips.
The 4 figures below show an example of REAPER MIDI setup[123].

[123] Note that you may have more than one set of WING MIDI remote control, 1, 2, and 3 showing depending on your configuration, or depending on the system state at last reboot or MIDI drivers enable state.



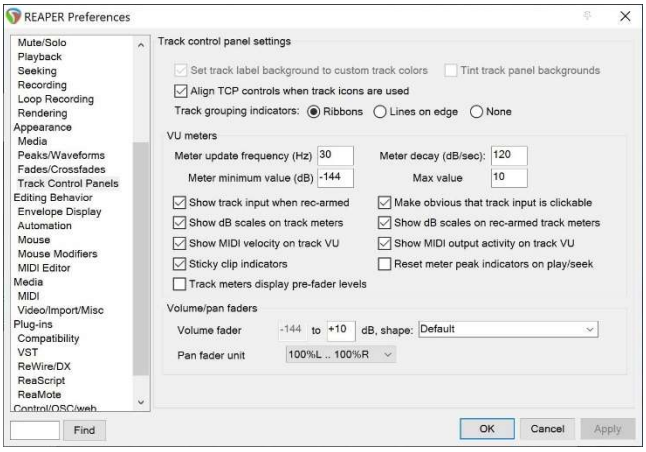
Note the "`Surface offset`" changes as we set `MCU`, `MCE #1` and `MCE #2`





: If the "ignore global bank offsets" flags are **not** checked in the REAPER MIDI surface control setup panels above, using the ◀ 4 and 4 ▶ WING buttons will enable you to navigate left and right in the REAPER tracks if more than 24 REAPER tracks are available.
The current global start index is shown at the top left of the DAW transport scribbles ('01" circled in red below)

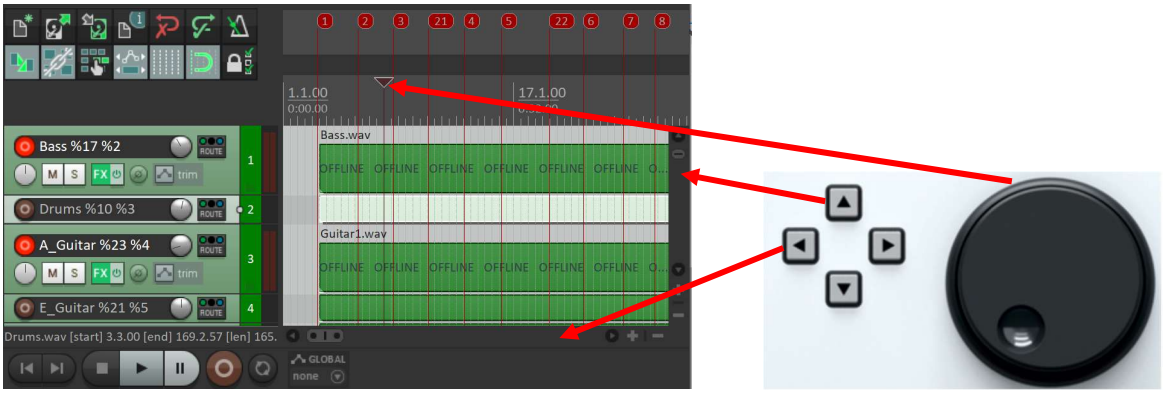
One last setting in REAPER consists in setting the fader scale to values matching WING -144dB→10dB faders. This can be done in the `Options→Preferences→Track Control Settings` panel by setting the min `Volume fader` to `-144dB`, the `max` to `+10dB` and selecting a `shape` type of `Default`, as presented below:



With the settings above, and WING DAW mode setup to `USB MIDI`, `MUC+2xExtenders`, you now have a 24 channel strips DAW surface control to manage REAPER tracks (`Volume, Solo, Select, Mute`) from your WING, and vice-versa (i.e. changes made on a surface (WING or REAPER) will reflect on the other);
REAPER tracks' `Pan` control can be achieved using the 4 rotary knobs in the WING control zone, situated just below the "Custom Controls" silkscreened text.
The `UP` and `DOWN` buttons on the left of the control zone can be used to navigate within REAPER strips as they are mapped to the WING surface strips.

REAPER tracks Rec/arm is possible using the buttons from the lower row of buttons in the WING control zone. The picture below shows Bass and A_Guit being armed for recording:

The `4 directional keys` located on the left of the `Jog wheel` will navigate into the REAPER audio window (i.e. identical to moving the audio window elevators).
The WING `Jog wheel` will move the REAPER audio cursor left and right (sometimes with a slight lag), and if the `Play` and `Scrub` buttons are simultaneously active, moving the wheel will scrub through audio after a small timeout not moving the wheel.



Flip to the WING audio standard controls and move the WING Main fader(s) to get some audio out, remember all other active USB inputs should be set at `0dB`.
Flipping back to `DAW Remote` mode and press the `Play` button. REAPER will start playing audio; the audio signal will flow to WING using USB/ASIO drivers, and will be managed by the WING audio engines. The faders on WING (in control surface mode) act as remote controls to REAPER track faders (and vice-versa) using USB MIDI.
All set!

Important note on USB & MIDI: Changing the clock rate or number of USB Audio channels on WING causes USB to disconnect for a few seconds (including MIDI). On certain operating systems, this may also reset already active MIDI connections. This could happen when loading snapshots with different clock rate or USB Audio configuration.

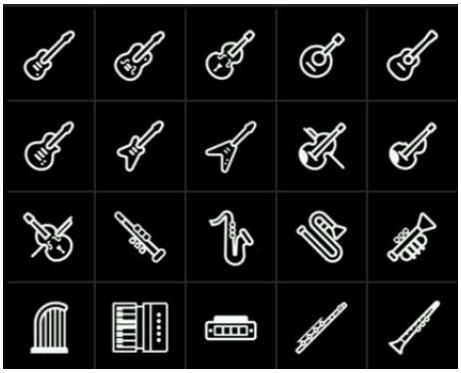
Appendix: WING Icons

The table below gives the list of icons available with WING. Icon number ranges are listed to the right of the icons.

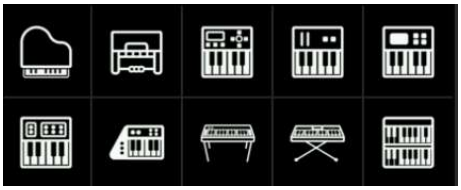
General:
[0…14]

Vocals and Mics:
[100…114]

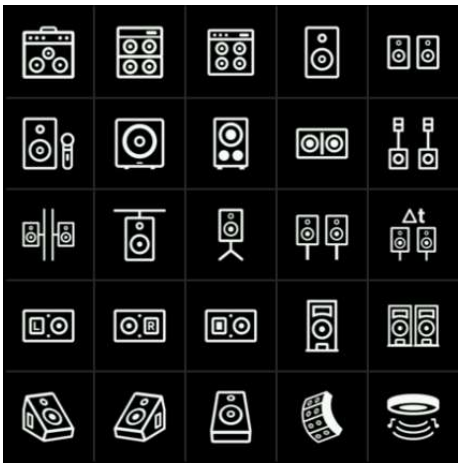
Drums and Percussions:
[200…224]



Strings and Winds:
[300…319]



Keys:
[400…409]

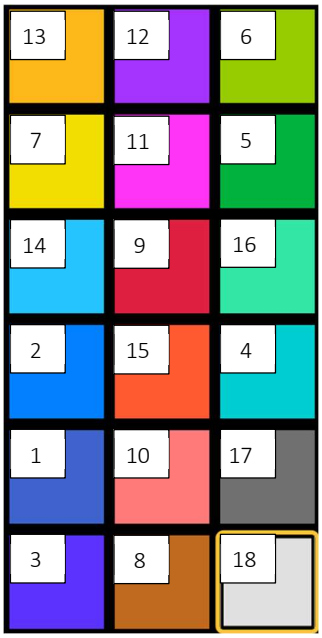


Speakers:
[500…524]

Specials:
[600…614]

Appendix: WING Colors

WING colors are used in several areas such as channel strip color, scribble color, etc. The known colors are shown below and indexed as values 1 to 18:



Appendix: WING GPIOs:

Description

The WING digital mixing console is offering 4 GPIOs (General Purpose Input/Output)[124] which can be very useful in the studio or live situations. This paragraph shows how to use them in different modes. Let's look at what GPIOs can offer.

At the rear of the console, two TRS jack sockets provide connections to 4 GPIOS. Each of the TRS sockets is depicted below. Lug L3 is common to the 2 GPIOs supported by each socket. Lugs L1 and L2 are respectively used for GPIO 1 or A, 2 or B or 3 or C, 4 or D, depending on the socket used.

WING GPIO 'mode' settings can be any of the following: `TGLNO, TGLNC, INNO, INNC, OUTNO, OUTNC`. These are represented by OSC patterns `/$ctl/gpio/1..4/mode`, and correspond to:

`TGLNO`	Toggle, Normally Opened
`TGLNC`	Toggle, Normally Closed
`INNO`	Input, Normally Opened
`INNC`	Input, Normally Closed
`OUTNO`	Output, Normally Opened
`OUTNC`	Output, Normally Closed

WING GPIO 'state' values can be 0 for `open/OFF` (light off), or 1 for `close/ON` (light on). These correspond to OSC patterns `/$ctl/gpio/1..4/gpstate`.

Electrical connections

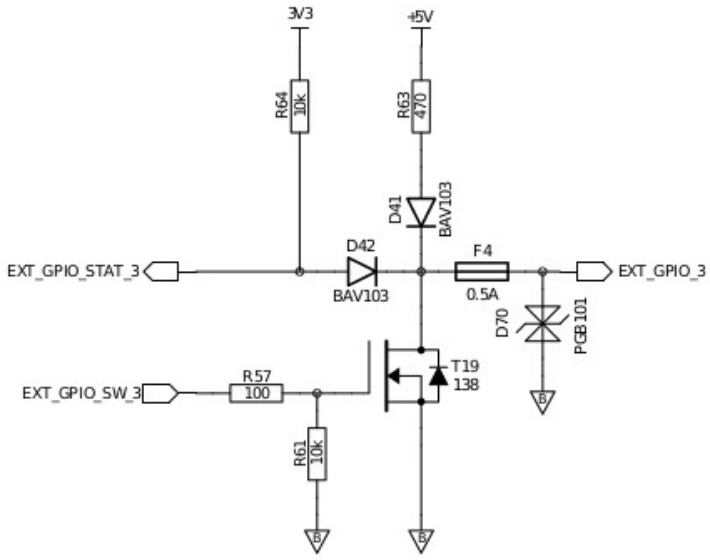
- `INNO / INNC`: The console provides approx. 5V between A/B/C/D and Common. The application of a short, dropping voltage to 0V will change the state of the respective GPIO between `open` and `close`, depending on the `NO/NC` mode.

- `OUTNO / OUTNC`: The console provides approx. 5V between A/B/C/D and Common; The voltage presented by the console goes from near 5V to 0V depending on the state (`open` or `close`) and the `NO/NC` mode of the respective GPIO.

[124] 2 GPIOs for the Compact model.

- `TGLNO / TGLNC`: This is to toggle the internal state of the GPIO. The console provides approx.. 5V between A/B/C/D and Common; changing the state of the respective GPIO does not change the voltage provided by the console.

As a general statement, care should be taken when connecting external devices to console. A partial circuit of GPIO implementation is shown here for reference:



From the schematic above, we can see a 470 Ohm resistor in series with a diode and a 500mA poly-fuse going to the GPIO port. A max current of approx. 100mA is therefore available for connecting a small relay or an LED. TVS diode D70 protects against electrostatic discharge (ESD).

Power-on delay

Wing GPIOs can be set to provide a one-time Power-on delay of up to 30s that can be used with `OUTNO / OUTNC` modes. This can be quite useful when one needs to power sync external gear with the console.

Note nevertheless GPIOs will always turn `ON` for a short period of time while powering on and booting up, before getting to their respective programmed state and a possible delay applies.

GPIO precedence on USER/LAYER CC GPIO function

The programming of GPIO takes precedence on a possible USER CC GPIO setup; If for example you have set GPIO 1/A to MODE `OUT-NC` and with a FLAG set to `DELAY 10S` for example, a further USER/LAYER CC button programming with FUNCTION set to `GPIO` will have no effect. If GPIO 1/A is programmed to MODE `OUT-NC` and has a FLAG set to `A-TOGGLE` or `A-PUSH` for example, then a further USER/LAYER CC button programming with FUNCTION set to `GPIO` will work when pressing the respective button as toggle or push, as programmed for the USER/LAYER CC button (i.e. GPIO 1/A can be FLAG `A-TOGGLE` and work as `A-PUSH` if the USER/LAYER CC button is set so GPIO 1/A can be FLAG `A-PUSH` and work as `A-TOGGLE` if the USER/LAYER CC button is set so).

Multiple, simultaneous actions, using GPIOs

Thanks to a clever use of GPIOs, it is possible to manage simultaneous or temporary/fugitive actions on WING and without the help from external applications[125], by assigning different actions to separate GPIOs, and then electrically attach said GPIOs to a single switch. When actioning the switch, the multiple GPIOs will become active or unactive (depending on their individual setting) and will carry on with their attached action on WING. Using this, it is possible from a single footswitch to load up to 4 separate snippets, or to mute a set of channels while unmuting others, or any combination that will help you in your mixing routines.
There are drawbacks though, such as the limit to the number of simultaneous actions that can be achieved and the impossibility to set two actions (one for `ON` state and a different one for `OFF` state) per GPIO.

[125] wcc or wxfade (https://x32ram.com/products) are example of applications that enable multiple simultaneous or temporary/fugitive actions, from a CC button/encoder, MIDI command, or GPIO.

Appendix: W-Live/SD card Sessions

Recording data format

The SD-card recording format is optimized for write speed ensuring long 32 channel recordings of 48 kHz / 32-bit PCM data, with minimal risk for audio drop-outs on a large variety of SD or SDHC cards. Class-10 cards (guaranteed 10MB/s write speed) are recommended.

To achieve optimum write performance, all tracks (8, 16 or 32) are written into a single file. The file format is 32-bit PCM multi-channel WAV. The supported card file system is **FAT32** (royalty free) thus limiting file sizes to 4GB.

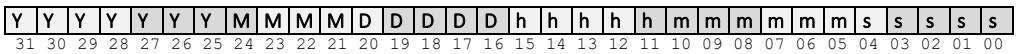
Recording 32 tracks of 48 kHz uncompressed 32-bit audio requires 360MB of memory/storage per minute. Hence, a 4GB file will hold less than 11.9 minutes at maximum audio bitrate, taking into account the necessary file header. To allow for longer consistent recording time, WING creates a so-called Session (i.e. a folder) containing one or more files (or *takes*), each file being up to 4GB in size.

Separating recorded sessions into individual wave files, or creating individual audio stems for playback require the use of external utilities[126] or can be managed directly from most DAW software (for separating sessions info individual files).

Session name coding

Recording a session on an SD card with WING will automatically create a subfolder underneath "X-LIVE", named by the 32-bit timestamp of the recording start as an 8-character hex-string, e.g. "4ACE72B1". The Console will read the folder name and display the corresponding timestamp as the session name, unless it was given another name (see below).

Session name (a timestamp) coding is done on 32 bits, and is represented by a string of 8 hexadecimal characters. The format “Year-Month-Day-Hour-Minute-Second” is detailed below:



Years are counted starting at 1980
Seconds are divided by 2

[126] **Live Sessions** (https://www.behringer.com/product.html?modelCode=0603-AEN) in the software downloads,
Live2Wav and **Wav2Live** (https://sites.google.com/site/patrickmaillot/x-m-w-live),
or **Wave Agent** (https://www.sounddevices.com/product/wave-agent-software/) are applications that provide both ways conversions.
Live SD Splitter (https://sites.google.com/view/x32-stuff-here/home) provides splitting capability.

Naming & sorting your existing sessions

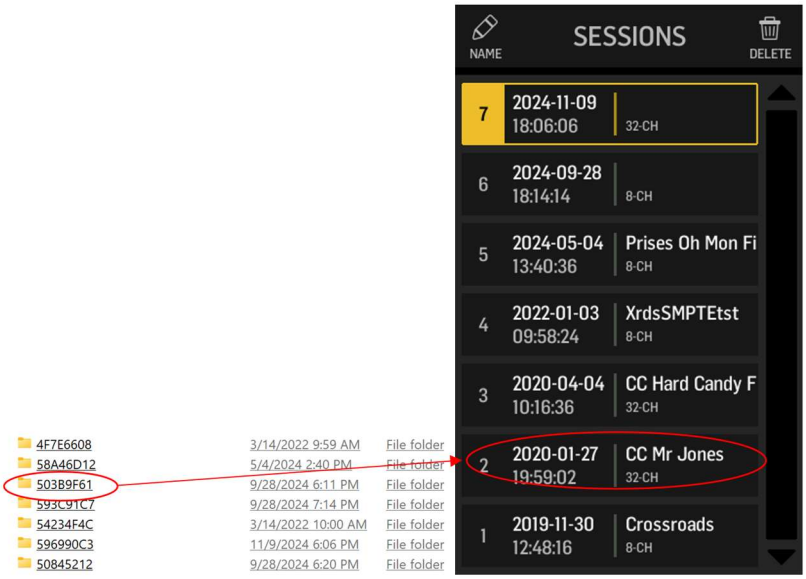
Unless you rename them or give them a user-friendly name, your recorded sessions will only display their creation date and the number of channels the recording is made of.

You can rename or provide a user-friendly name to your recorded sessions with using the 

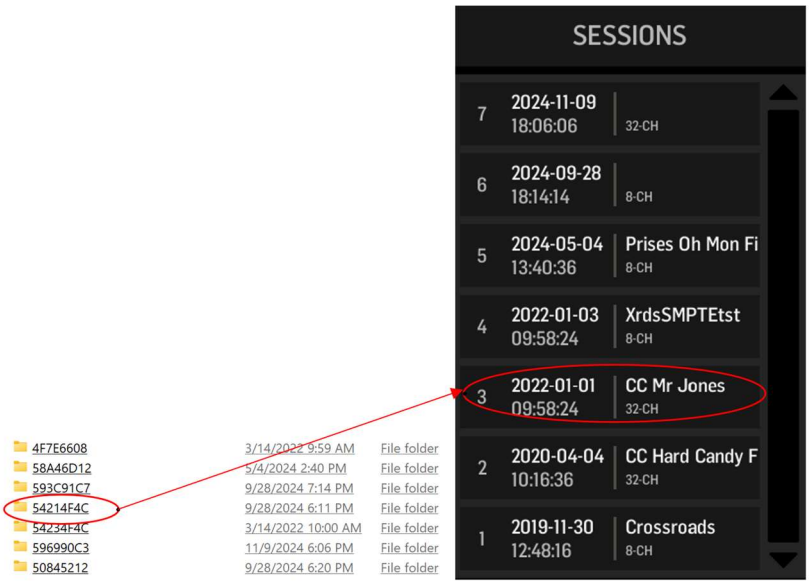
option on the WING SD card screen. This information is saved in the `SE_LOG.BIN` file in the session directory and will not change the session name per-se; It is used by WING to correctly display your recording name.

WING sessions are displayed and sorted by their creation date; You can change/set the order in which your recorded sessions will display by renaming your sessions using a PC for example, abut you must respect the hexadecimal format used for session naming.

The pictures below show the effect of changing the name of WING sessions on the displaying order of these.



By renaming Session 503B9F61 into 54214F4C, we change the timestamp change for “CC Mr Jones”, resulting in a change in the songs displaying order:



While effective, the above implies recoding timestamps and results in the loss of the original timestamp for your SD recordings. It may or may not fit with your needs, and using an external application[127] for listing, playing or selecting SD sessions may be a better approach.

[127] Such as **wplayer** (https://sites.google.com/site/patrickmaillot/wing#h.asfjltealgzl) for example

Working with Dante or WSG

Wing offers the capability to use Dante or WSG for connecting to, routing, and transporting audio.

- Dante is the product name for a combination of software, hardware, and network protocols that delivers uncompressed, multi-channel, low-latency digital audio over a standard Ethernet network using Layer 3 IP packets. Developed in 2006 by Audinate, Dante builds on previous audio over Ethernet and audio over IP technologies.[128]

- WSG (Wave SoundGrid) is a networking and processing platform audio application made by Waves Audio and developed in cooperation with DiGiCo. It consists of a Linux-based server that runs the SoundGrid environment, compatible plug-ins, a Mac or Windows control computer, and an audio interface for input/output (I/O). It provides a low-latency environment for audio processing on certain hardware audio mixing consoles.[129]

Wing add-on options can be an extension card (W-DANTE) that replaces the standard SD card, or an internal module (Dante or WSG) that can be directly installed in a dedicated module slot on the main motherboard of the console; While it is recommended to have this operation performed by a trained technician, end-users can perform the installation of an internal module if carefully following simple rules and steps shown in a Music Tribe video[130].
There is a clear advantage in using internal modules as these offer additional sources and routing options to the console without scarifying the SD recording/playing capabilities Wing offers out of the box.

Dante or WSG offer a typical additional 64 INs and 64 OUTs to the console and opens routing and audio treatment capabilities to a very large set of devices proposed by many audio devices manufacturers. Please refer to the respective manufacturers' websites for a complete list and audio management capabilities.

Dante modules (card supported or internal) must have a FW that matches with the Wing's internal FW, so it is important to keep your gear up to date. The unofficial Behringer.world[131] forum has a wiki section on how to perform updates of the Dante modules.

With the module or extension card installed, Wing offers more routing options to digital audio sources or destinations on an existing 1Gb/s network as part of its routing options under the `DANTE` or `WSG` icon. Please note it will likely be necessary to set the `SETUP→AUDIO→AUDIO CLOCK Sync Source` to the installed module so you ensure proper synchronization with other devices on the network. If not doing so, you will likely encounter issues in sending or receiving digital audio from and to the network.

[128] Source: Wikipedia
[129] Source: Wikipedia
[130] https://www.youtube.com/watch?v=B9z42_4HOp8
[131] https://behringer.world/wiki/doku.php?id=flash_wing_dante

Appendix: MCU [DAW BUTTONS] commands list

OSC	MCU action	MIDI (port 4)		OSC	MCU action	MIDI (port 4)
T1	STOP	90, 5D, 7F/00		V7	BUSES (VIEW)	90, 43, 7F/00
T2	PLAY	90, 5E, 7F/00		V8	OUTPUTS (VIEW)	90, 44, 7F/00
T3	RECORD	90, 5F, 7F/00		V9	USER (VIEW)	90, 45, 7F/00
T4	REWIND	90, 5B, 7F/00		V10	MIX (VIEW)	
T5	FAST FWD	90, 5C, 7F/00		V11	EDIT (VIEW)	
T6	MARKER	90, 54, 7F/00		V12	TRANSPORT (VIEW)	
T7	NUDGE	90, 55, 7F/00		V13	MEM/LOC (VIEW)	
T8	CYCLE	90, 56, 7F/00		V14	STATUS (VIEW)	
T9	DROP	90, 57, 7F/00		V15	ALT (VIEW)	
T10	REPLACE	90, 58, 7F/00		AU1	READ/OFF (AUTOM)	90, 4A, 7F/00
T11	SCRUB	90, 65, 7F/00		AU2	WRITE (AUTOM)	90, 4B, 7F/00
T12	SHUTTLE			AU3	TRIM (AUTOM)	90, 4C, 7F/00
T13	RETURN TO ZERO			AU4	TOUCH (AUTOM)	90, 4D, 7F/00
T14	GO TO END			AU5	LATCH (AUTOM)	90, 4E, 7F/00
T15	IN			AU6	OFF (AUTOM)	
T16	OUT			AU7	FADER (AUTOM)	
T17	PRE			AU8	PAN (AUTOM)	
T18	POST			AU9	MUTE (AUTOM)	
T19	ONLINE			AU10	SEND (AUTOM)	
T20	QUICK PUNCH			AU11	SEND MUTE (AUTOM)	
N1	UP (NAV)	90, 60, 7F/00		AU12	PLUG-IN (AUTOM)	
N2	DOWN (NAV)	90, 61, 7F/00		SY1	SHIFT	90, 46, 7F/00
N3	LEFT (NAV)	90, 62, 7F/00		SY2	OPTION	90, 47, 7F/00
N4	RIGHT (NAV)	90, 63, 7F/00		SY3	CTRL	90, 48, 7F/00
N5	ZOOM	90, 64, 7F/00		SY4	ALT	90, 49, 7F/00
N6	BK <	90, 2E, 7F/00		SY5	SAVE	90, 50, 7F/00
N7	BK >	90, 2F, 7F/00		SY6	UNDO	90, 51, 7F/00
N8	CH <	90, 30, 7F/00		SY7	CANCEL	90, 52, 7F/00
N9	CH >	90, 31, 7F/00		SY8	ENTER	90, 53, 7F/00
A1	TRACK (ASSIGN)	90, 28, 7F/00		SY9	EDIT MODE	
A2	SEND (ASSIGN)	90, 29, 7F/00		SY10	EDIT TOOL	
A3	PAN (ASSIGN)	90, 2A, 7F/00		OT1	FLIP	90, 32, 7F/00
A4	PLUG-IN (ASSIGN)	90, 2B, 7F/00		OT2	GROUP	90, 4F, 7F/00
A5	EQ (ASSIGN)	90, 2C, 7F/00		OT3	NAME/VALUE	90, 34, 7F/00
A6	INST (ASSIGN)	90, 2D, 7F/00		OT4	TIME/BEATS	90, 35, 7F/00
A7	SEND A (ASSIGN)			OT5	CLICK	90, 59, 7F/00
A8	SEND B (ASSIGN)			OT6	SOLO	90, 5A, 7F/00
A9	SEND C (ASSIGN)			OT7	FOOTSW A	90, 66, 7F/00
A10	SEND D (ASSIGN)			OT8	FOOTSW B	90, 67, 7F/00
A11	SEND E (ASSIGN)			OT9	DEFAULT	
A12	INPUT (ASSIGN)			OT10	SUSPEND	
A13	OUTPUT (ASSIGN)			OT11	BYPASS	
A14	ASSIGN (ASSIGN)			OT12	RECRDY ALL	
A15	SHIFT (ASSIGN)			E1	CUT (EDIT)	
A16	MUTE (ASSIGN)			E2	COPY (EDIT)	
F1	F1	90, 36, 7F/00		E3	PASTE (EDIT)	
F2	F2	90, 37, 7F/00		E4	SEPARATE (EDIT)	
F3	F3	90, 38, 7F/00		E5	CAPTURE (EDIT)	
F4	F4	90, 39, 7F/00		E6	DELETE (EDIT)	
F5	F5	90, 3A, 7F/00		E7	ASSIGN (EDIT)	
F6	F6	90, 3B, 7F/00		E8	COMPARE (EDIT)	
F7	F7	90, 3C, 7F/00		E9	BYPASS (EDIT)	
F8	F8	90, 3D, 7F/00		E10	INS/PARAM (EDIT)	
V1	GLOBAL (VIEW)	90, 33, 7F/00		SP1	FADER TOUCH [MUTE]	
V2	MIDI (VIEW)	90, 3E, 7F/00		SP2	V-POT CTRL [SEL/SOLO]	
V3	INPUTS (VIEW)	90, 3F, 7F/00		SP3	RECRDY CTRL [SEL]	
V4	AUDIO TRACKS (VIEW)	90, 40, 7F/00		SP4	AUTO [SEL]	
V5	INSTRUMENT (VIEW)	90, 41, 7F/00		SP5	V-SEL [SEL]	

V6	AUX (VIEW)	90, 42, 7F/00		SP6	INSERT [SEL]	

Appendix: MCU [DAW V-POTS] commands list

OSC	MCU action	MIDI (port 4)		OSC	MCU action	MIDI (port 4)
M1P	V-POT M1 Push	90, 20, 7F/00		M1	V-POT M1	B0, 10, 01/41
M2P	V-POT M2 Push	90, 21, 7F/00		M2	V-POT M2	B0, 11, 01/41
M3P	V-POT M3 Push	90, 22, 7F/00		M3	V-POT M3	B0, 12, 01/41
M4P	V-POT M4 Push	90, 23, 7F/00		M4	V-POT M4	B0, 13, 01/41
M5P	V-POT M5 Push	90, 24, 7F/00		M5	V-POT M5	B0, 14, 01/41
M6P	V-POT M6 Push	90, 25, 7F/00		M6	V-POT M6	B0, 15, 01/41
M7P	V-POT M7 Push	90, 26, 7F/00		M7	V-POT M7	B0, 16, 01/41
M8P	V-POT M8 Push	90, 27, 7F/00		M8	V-POT M8	B0, 17, 01/41
E1P	V-POT EXT1 Push			E1	V-POT EXT1	
E2P	V-POT EXT2 Push			E2	V-POT EXT2	
E3P	V-POT EXT3 Push			E3	V-POT EXT3	
E4P	V-POT EXT4 Push			E4	V-POT EXT4	
E5P	V-POT EXT5 Push			E5	V-POT EXT5	
E6P	V-POT EXT6 Push			E6	V-POT EXT6	
E7P	V-POT EXT7 Push			E7	V-POT EXT7	
E8P	V-POT EXT8 Push			E8	V-POT EXT8	
E9P	V-POT EXT9 Push			E9	V-POT EXT9	
E10P	V-POT EXT10 Push			E10	V-POT EXT10	
E11P	V-POT EXT11 Push			E11	V-POT EXT11	
E12P	V-POT EXT12 Push			E12	V-POT EXT12	
E13P	V-POT EXT13 Push			E13	V-POT EXT13	
E14P	V-POT EXT14 Push			E14	V-POT EXT14	
E15P	V-POT EXT15 Push			E15	V-POT EXT15	
E16P	V-POT EXT16 Push			E16	V-POT EXT16	
				JOG	JOG WHEEL	B0, 3C, 01/41

Appendix: MCU [DAW REMOTE MCU] commands list

OSC	MCU action	MIDI (port 4)
M1	V-POT M1	B0, 10, 01/41
M2	V-POT M2	B0, 11, 01/41
M3	V-POT M3	B0, 12, 01/41
M4	V-POT M4	B0, 13, 01/41
M5	V-POT M5	B0, 14, 01/41
M6	V-POT M6	B0, 15, 01/41
M7	V-POT M7	B0, 16, 01/41
M8	V-POT M8	B0, 17, 01/41
E1	V-POT EXT1	
E2	V-POT EXT2	
E3	V-POT EXT3	
E4	V-POT EXT4	
E5	V-POT EXT5	
E6	V-POT EXT6	
E7	V-POT EXT7	
E8	V-POT EXT8	
E9	V-POT EXT9	
E10	V-POT EXT10	
E11	V-POT EXT11	
E12	V-POT EXT12	
E13	V-POT EXT13	
E14	V-POT EXT14	
E15	V-POT EXT15	
E16	V-POT EXT16	
JOG	JOG WHEEL	B0, 3C, 01/41

Appendix: WING Snapshot and JSON Data Structure:

A WING snapshot file (also called Snapfile when saved to a file) is organized as a collection of classes, sub-classes and objects regrouping attributes and values in logical groups. These can be represented as a hierarchical tree. A `JSON`[132] notation is used to describe and store the hierarchical tree.
A complete WING default snapfile is close to 800000 bytes and 33895 lines, containing a rather complex hierarchical list of more than 30000 object identifiers [WING parameters] and their associated values.
A WING snapfile does not contain read-only objects; i.e. there are more parameters available than the ones listed/saved in a snapfile!

Wing Snapfile

A snapfile is divided in sections as shown below:


```
{
  "type": "snapshot.9",
  "creator_fw": "2.1-117-gbfb74b95:develop",
  "creator_sn": "NO_SERIAL",
  "creator_model": "wing",
  "creator_version": "SX45-XU2",
  "creator_name": "HMS-01",
  "created": "2024-08-02 12:58:09",
  "active_show": "",
  "active_scene": "I:/FOLDER 1/SC0-NEW.snap",
  "ae_data": {
  "ce_data": {
  "ae_globals": {
  "ce_globals": {
  "scopes": {
  "updated": "2024-08-02 13:05:34"
}
```


`ae_data` and `ce-data` are representing WING Audio engine and Console Engine data
`ae_globals` and `ce-globals` consist of global data affecting WING Audio and Console Engines

Description

description: This small section contains (as its name suggest) a description for the snapshot, including name, and elements corresponding to the WING that generated the snapshot. “created” lists the date and time of the creation of the snapfile, while “updated” will retain the date and time of the most recent update made to the file.

```
“type”: string, snapshot signature/version
“creator_fw”: string, FW used when creating the snapshot
“creator_sn”: string, Serial number of the WING the snapshot was created with
“creator_model”: string, Model of console
“creator_name”: string, Name given to the console
“created”: string, date time,
“active_show”: string, name of the currently opened show file
“active_scene”: string, name of the current, active scene
“updated”: string, date time,
```

[132] JavaScript Object Notation: an efficient way to represent structured objects. Also used as a data-interchange format.

scopes

scopes: A large set of *BooLean* {***'+',' '***} values to list what has been 'marked' at snapshot time. This can be used as a reminder of the initial purpose of the snapshot.
The scopes class contains the following objects:

ch, aux, bus, main, mtx, dca, mute, fx, source, output, area, custom, setup, contents, mainsend, bussend;

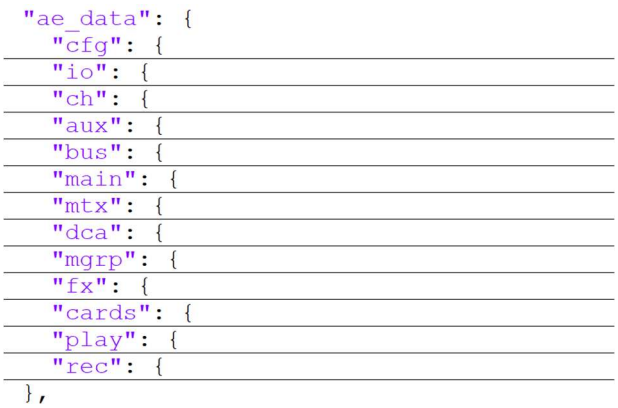
For example:


```
"scopes": {
  "ch": "++++++++++++++++++++++++++++++++++++",
  "aux": "++++++++",
  "bus": "++++++++++++++++",
  "main": "++++",
  "mtx": "++++++++",
  "dca": "++++++++++++++++",
  "mute": "++++++++",
  "fx": "++++++++++++++++",
  "source": {
    "LCL": "++++++++++++++++++++++",
    "AUX": "++++++++",
    "A": "++++++++++++++++++++++++++++++++++++++++++++",
    "B": "++++++++++++++++++++++++++++++++++++++++++++",
    "C": "++++++++++++++++++++++++++++++++++++++++++++",
    "SC": "++++++++++++++++++++++++++++++++",
    "USB": "++++++++++++++++++++++++++++++++++++++++++++",
    "CRD": "++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++",
    "MOD": "++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++",
    "PLAY": "++++",
    "AES": "++",
    "USR": "++++++++++++++++++++++++++++++++++++++++++++",
    "OSC": "++"
  },
  "output": {
    "LCL": "++++++++",
    "AUX": "++++++++",
    "A": "++++++++++++++++++++++++++++++++++++++++++++",
    "B": "++++++++++++++++++++++++++++++++++++++++++++",
    "C": "++++++++++++++++++++++++++++++++++++++++++++",
    "SC": "++++++++++++++++++++++++++++++++",
    "USB": "++++++++++++++++++++++++++++++++++++++++++++",
    "CRD": "++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++",
    "MOD": "++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++",
    "REC": "++++",
    "AES": "++"
  },
  "area": {
    "LEFT": "+++++++++",
    "CENTER": "+++++++++",
    "RIGHT": "+++++++++"
  },
  "custom": "++++++++++++++++++++++",
  "setup": "+++",
  "contents": "++++++++++++++++",
  "mainsend": "++++",
  "bussend": "++++++++++++++++++++++++"
}
```

Scopes are not elements that can be programmed/changed. They are only set at snapshot time using the console main LCD. As mentioned above, they are optionally saved at save time to notify what was targeted for save/update.

ae_data

ae_data stands for "Audio Engine", and regroups a rather large set of attributes and values aimed at registering all main settings of the WING audio engine, such as Routing, Channel EQ settings, FX parameter values, etc., as shown in the figure below:



Expanding one (or any) of the blocks of parameters listed above will provide the respective WING parameters of that block, along with their value(s). Understanding what parameters are present in each block is a good way to better grasp and understand the vast range of capabilities WING offers. It is also a good way to envision the parameter list one can get and set using **`wapi`** (described earlier in this document) as the `JSON` structure parameters matches the tokens used by the API for **`wapi`** `get()` and `set()` functions.

Indeed, all tokens related to the audio engine can be directly coded from the `JSON` description, for example, the C-like token notation for the `JSON` ***`cfg.mon.1.pan`*** element is named `CFG_MON_1_PAN`.