mod_commands

- See Also

---

0. About

`mod_commands` processes the FreeSWITCH API commands.

FreeSWITCH API

The public FreeSWITCH API consists of all the commands that can be issued to FreeSWITCH via

- its console, `fs_cli`,
- the event socket interface, and
- scripting interfaces.

> The set of available commands is dependent on which modules are loaded. The authoritative set of commands for your installation is the union of the sets of commands registered by each module.
>
> To see a list of available API commands simply type **`help`** or **`show api`** at the CLI.

1. Usage

1.1 CLI

See below.

1.2 API/Event Interfaces

- mod_event_socket
- mod_erlang_event
- mod_xml_rpc

1.3 Scripting Interfaces

- mod_perl
- mod_v8
- mod_python
- mod_lua

1.4 From the Dialplan

An API command can be called from the dialplan. Example:

Invoke API Command From Dialplan

```
<extension name="Make API call from Dialplan">
  <condition field="destination_number" expression="^(999)$">
    <!-- next line calls hupall, so be careful! -->
    <action application="set" data="api_result=${hupall(normal_clearing)}"/>
  </condition>
</extension>
```

Other examples:

Other Dialplan API Command Examples

```
<action application="set" data="api_result=${status()}"/>
<action application="set" data="api_result=${version()}"/>
<action application="set" data="api_result=${strftime()}"/>
<action application="set" data="api_result=${expr(1+1)}"/>
```

API commands with multiple arguments usually have the arguments separated by a space:

Multiple Arguments

```
<action application="set" data="api_result=${sched_api(+5 none avmd ${uuid} start)}"/>
```


Dialplan Usage

If you are calling an API command from the dialplan make absolutely certain that there isn't already a dialplan application that gives you the functionality you are looking for. See mod_dptools for a list of dialplan applications, they are quite extensive.

2. Format of returned data

Results of some status and listing commands are presented in comma delimited lists by default.

Data returned from some modules may also contain commas, making it difficult to automate result processing. They may be able to be retrieved as

- XML by appending the string **`as xml`**
- JSON by appending the string **`as json`**
- the default format but changing the default comma delimiter by appending **`as delim <custom_delimiter>`**

to the end of the command string.

3. Core Commands

Extraction script

Mitch Capper wrote a Perl script to extract commands from `mod_commands`'s source code, `mod_commands.c` , but should work for most other files as well.

Extraction Perl Script

```
#!/usr/bin/perl
use strict;
open (fl,"src/mod/applications/mod_commands/mod_commands.c");
my $cont;
{
        local $/ = undef;
        $cont = <fl>;
}
close fl;
my %DEFINES;
my $reg_define = qr/[A-Za-z0-9_]+/;
my $reg_function = qr/[A-Za-z0-9_]+/;
my $reg_string_or_define = qr/(?:(?:$reg_define)|(?:"[^"]*"))/;

#load defines
while ($cont =~ /
                                        ^\s* \#define \s+ ($reg_define) \s+ \"([^"]*)\"
                                /mgx){
        warn "$1 is #defined multiple times" if ($DEFINES{$1});
        $DEFINES{$1} = $2;
}

sub resolve_str_or_define($){
        my ($str) = @_;
        if ($str =~ s/^"// && $str =~ s/"$//){ #if starts and ends with a quote strip them off and
return the str
                return $str;
        }
        warn "Unable to resolve define: $str" if (! $DEFINES{$str});
        return $DEFINES{$str};
}
#parse commands
while ($cont =~ /
                                        SWITCH_ADD_API \s* \( ([^,]+) #interface $1
                                        ,\s* ($reg_string_or_define) # command $2
                                        ,\s* ($reg_string_or_define) # command description $3
                                        ,\s* ($reg_function) # function $4
                                        ,\s* ($reg_string_or_define) # usage $5
                                        \s*\);
                                /sgx){
                my ($interface,$command,$descr,$function,$usage) = ($1,$2,$3,$4,$5);
                $command = resolve_str_or_define($command);
                $descr = resolve_str_or_define($descr);
                $usage = resolve_str_or_define($usage);
                warn "Found a not command interface of: $interface for command: $command" if
($interface ne "commands_api_interface");
                print "$command -- $descr -- $usage\n";
}
```

3.1 `acl`

Match an IP address against an access control list (ACL).

3.1.1 Syntax

Syntax

```
acl <ip_address> <acl_name>
```

3.1.2 Examples

Examples

```
acl 1.2.3.4 test
```

3.2 `alias`

Provide an alternative name (i.e., an alias) to commonly used commands on the CLI to save on some keystrokes.

Subcommand	Description
add	Create an alias.
stickyadd	Create an alias that persists across restarts.
del	Delete alias.

3.2.1 Syntax

Syntax

```
alias add <alias> <command(s)>

alias stickyadd <alias> <command(s)>

alias del [<alias>|*]
```

3.2.2 Examples

Examples

```
freeswitch> alias add reloadall reloadacl reloadxml
+OK

freeswitch> alias add unreg sofia profile internal flush_inbound_reg
+OK

freeswitch> alias stickyadd reloadall reloadacl reloadxml
+OK

freeswitch> alias del reloadall
+OK
```

3.3 `bgapi`

Execute an API command in a thread.

3.3.1 Syntax

Syntax

```
bgapi <command>[ <command_args>]
```

3.3.2 Examples

<table>
<tr><th>Command</th><th>Description</th><th>Syntax</th><th>Examples</th></tr>
<tr><td>acl</td><td>Match an IP address against an access control list (ACL).</td><td>acl <ip_address> <acl_name></td><td>acl 1.2.3.4 test</td></tr>
<tr><td>alias</td><td>Provide an alternative name (i.e., an alias) to commonly used commands on the CLI to save on some keystrokes.
<table><tr><th>Subcommand</th><th>Description</th></tr><tr><td>add</td><td>Create an alias.</td></tr><tr><td>stickyadd</td><td>Create an alias that persists across restarts.</td></tr><tr><td>del</td><td>Delete alias.</td></tr></table></td><td><code>alias add <alias> <command(s)></code>

<code>alias stickyadd <alias> <command(s)></code>

<code>alias del [<alias>|*]</code></td><td>freeswitch> alias add reloadall reloadacl reloadxml
+OK

freeswitch> alias add unreg sofia profile internal flush_inbound_reg
+OK

freeswitch> alias stickyadd reloadall reloadacl reloadxml
+OK

freeswitch> alias del reloadall
+OK</td></tr>
<tr><td>bgapi</td><td>Execute an API command in a thread.</td><td>bgapi <command>[<command_args>]</td><td></td></tr>
<tr><td>complete</td><td>Complete.

TODO That description is not very helpful.</td><td>complete add <word>|del [<word>|*]</td><td></td></tr>
<tr><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td></tr>
</table>

`complete`

Complete.

`Usage: complete add <word>|del [<word>|*]`

cond

Evaluate a conditional expression.

`Usage: cond <expr> ? <true val> : <false val>`

Operators supported by `<expr>` are:

- == (equal to)
- != (not equal to)
- > (greater than)
- >= (greater than or equal to)
- < (less than)
- <= (less than or equal to)

How are values compared?

- two strings are compared as strings
- two numbers are compared as numbers
- a string and a number are compared as `strlen(string)` and numbers

For example, `foo == 3` evaluates to true, and `foo == three` to false.

Example:

Return true if first value is greater than the second

```
cond 5 > 3 ? true : false
true
```

Example in dialplan:

```
<action application="set" data="voicemail_authorized=${cond(${sip_authorized} == true ? true : false)}"/>
```

Slightly more complex example:

```
<action application="set" data="voicemail_authorized=${cond(${sip_acl_authed_by} == domains ? false :
${cond(${sip_authorized} == true ? true : false)})}"/>
```

Note about syntax

The whitespace around the question mark and colon are required since FS-5945. Before that, they were optional. If the spaces are missing, the cond function will return -ERR.

domain_exists

Check if a FreeSWITCH domain exists.

```
Usage: domain_exists <domain>
```

eval

Eval (noop). Evaluates a string, expands variables. Those variables that are set only during a call session require the uuid of the desired session or else return "-ERR no reply".

```
Usage: eval [uuid:<uuid> ]<expression>
```

Examples:

```
eval ${domain}
10.15.0.94

eval Hello, World!
Hello, World!

eval uuid:e72aff5c-6838-49a8-98fb-84c90ad840d9 ${channel-state}
CS_EXECUTE
```

expand

Execute an API command with variable expansion.

```
Usage: expand [uuid:<uuid> ]<cmd> <args>
```

Example:

```
expand originate sofia/internal/1001%${domain} 9999
```

In this example the value of ${domain} is expanded. If the domain were, for example, "192.168.1.1" then this command would be executed:

```
originate sofia/internal/1001%192.168.1.1 9999
```

fsctl

Send control messages to FreeSWITCH.

```
USAGE: fsctl
   [
   api_expansion [on|off] |
   calibrate_clock |
   debug_level [level] |
   debug_sql |
   default_dtmf_duration [n] |
   flush_db_handles |
   hupall |
   last_sps |
   loglevel [level] |
   max_dtmf_duration [n] |
   max_sessions [n] |
   min_dtmf_duration [n] |
   min_idle_cpu [d] |
   pause [inbound|outbound] |
   pause_check [inbound|outbound] |
   ready_check |
   reclaim_mem |
   recover |
   resume [inbound|outbound] |
   save_history |
   send_sighup |
   shutdown [cancel|elegant|asap|now|restart] |
   shutdown_check |
   sps |
   sps_peak_reset |
   sql [start] |
   sync_clock |
   sync_clock_when_idle |
   threaded_system_exec |
   verbose_events [on|off]
   ]
```

fsctl arguments

api_expansion

```
Usage: fsctl api_expansion [on|off]
```

Toggles API expansion. With it off, no API functions can be expanded inside channel variables like ${show channels} This is a specific security mode that is not often used.

calibrate_clock

```
Usage: fsctl calibrate_clock
```

Runs an algorithm to compute how long it actually must sleep in order to sleep for a true 1ms. It's only useful in older kernels that don't have timerfd. In those older kernels FS auto detects that it needs to do perform that computation. This command just repeats the calibration.

debug_level

```
Usage: fsctl debug_level [level]
```

Set the amount of debug information that will be posted to the log. 1 is less verbose while 9 is more verbose. Additional debug messages will be posted at the ALERT loglevel.

0 - fatal errors, panic

1 - critical errors, minimal progress at subsystem level

2 - non-critical errors

3 - warnings, progress messages

5 - signaling protocol actions (incoming packets, ...)

7 - media protocol actions (incoming packets, ...)

9 - entering/exiting functions, very verbatim progress

debug_sql

```
Usage: fsctl debug_sql
```

Toggle core SQL debugging messages on or off each time this command is invoked. Use with caution on busy systems. In order to see all messages issue the "logelevel debug" command on the fs_cli interface.

default_dtmf_duration

```
Usage: fsctl default_dtmf_duration [int]
```

int = number of clock ticks

Example:

```
fsctl default_dtmf_duration 2000
```

This example sets the default_dtmf_duration switch parameter to 250ms. The number is specified in clock ticks (CT) where duration (milliseconds) = CT / 8 or CT = duration * 8

The default_dtmf_duration specifies the DTMF duration to use on originated DTMF events or on events that are received without a duration specified. This value is bounded on the lower end by min_dtmf_duration and on the upper end by max_dtmf_duration. So max_dtmf_duration >= default_dtmf_duration >= min_dtmf_duration . This value can be set persistently in switch.conf.xml

To check the current value:

```
fsctl default_dtmf_duration 0
```

FS recognizes a duration of 0 as a status check. Instead of setting the value to 0, it simply returns the current value.

flush_db_handles

```
Usage: fsctl flush_db_handles
```

Flushes cached database handles from the core db handlers. FreeSWITCH reuses db handles whenever possible, but a heavily loaded FS system can accumulate a large number of db handles during peak periods while FS continues to allocate new db handles to service new requests in a FIFO manner. "fsctl flush_db_handles" closes db connections that are no longer needed to avoid exceeding connections to the database server.

hupall

```
Usage: fsctl hupall <clearing_type> dialed_ext <extension>
```

Disconnect existing calls to a destination and post a clearing cause.

For example, to kill an active call with normal clearing and the destination being extension 1000:

```
fsctl hupall normal_clearing dialed_ext 1000
```

last_sps

```
Usage: fsctl last_sps
```

Query the actual sessions-per-second.

```
fsctl last_sps
+OK last sessions per second: 723987253
```

(Your mileage might vary.)

loglevel

```
Usage: fsctl loglevel [level]
```

Filter much detail the log messages will contain when displayed on the fs_cli interface. See mod_console for legal values of "level" and further discussion.

The available loglevels can be specified by number or name:

```
0 - CONSOLE
1 - ALERT
2 - CRIT
3 - ERR
4 - WARNING
5 - NOTICE
6 - INFO
7 - DEBUG
```

max_sessions

```
Usage: fsctl max_sessions [int]
```

Set how many simultaneous call sessions FS will allow. This value can be ascertained by load testing, but is affected by processor speed and quantity, network and disk bandwidth, choice of codecs, and other factors. See switch.conf.xml for the persistent setting max-sessions.

max_dtmf_duration

```
Usage: fsctl max_dtmf_duration [int]
```

Default = 192000 clock ticks

Example:

```
fsctl max_dtmf_duration 80000
```

This example sets the max_dtmf_duration switch parameter to 10,000ms (10 seconds). The integer is specified in clock ticks (CT) where CT / 8 = ms. The max_dtmf_duration caps the playout of a DTMF event at the specified duration. Events exceeding this duration will be truncated to this duration. You cannot configure a duration that exceeds this setting. This setting can be lowered, but cannot exceed 192000 (the default). This setting cannot be set lower than min_dtmf_duration. This setting can be set persistently in switch.conf.xml as max-dtmf-duration.

To query the current value:

```
fsctl max_dtmf_duration 0
```

FreeSWITCH recognizes a duration of 0 as a status check. Instead of setting the value to 0, it simply returns the current value.

min_dtmf_duration

```
Usage: fsctl min_dtmf_duration [int]
```

Default = 400 clock ticks

Example:

```
fsctl min_dtmf_duration 800
```

This example sets the min_dtmf_duration switch parameter to 100ms. The integer is specified in clock ticks (CT) where CT / 8 = ms. The min_dtmf_duration specifies the minimum DTMF duration to use on outgoing events. Events shorter than this will be increased in duration to match min_dtmf_duration. You cannot configure a DTMF duration on a profile that is less than this setting. You may increase this value, but cannot set it lower than 400 (the default). This value cannot exceed max_dtmf_duration. This setting can be set persistently in switch.conf.xml as min-dtmf-duration.

It is worth noting that many devices squelch in-band DTMF when sending RFC 2833. Devices that squelch in-band DTMF have a certain reaction time and clamping time which can sometimes reach as high as 40ms, though most can do it in less than 20ms. As the shortness of your DTMF event duration approaches this clamping threshold, the risk of your DTMF being ignored as a squelched event increases. If your call is always IP-IP the entire route, this is likely not a concern. However, when your call is sent to the PSTN, the RFC 2833 DTMF events must be encoded in the audio stream. This means that other devices down the line (possibly a PBX or IVR that you are calling) might not hear DTMF tones that are long enough to decode and so will ignore them entirely. For this reason, it is recommended that you do not send DTMF events shorter than 80ms.

TODO RFC 2833 is obsoleted by RFC 4733.

Checking the current value:

```
fsctl min_dtmf_duration 0
```

FreeSWITCH recognizes a duration of 0 as a status check. Instead of setting the value to 0, it simply returns the current value.

min_idle_cpu

```
Usage: fsctl min_idle_cpu [int]
```

Allocates the minimum percentage of CPU idle time available to other processes to prevent FreeSWITCH from consuming all available CPU cycles.

Example:

```
fsctl min_idle_cpu 10
```

This allocates a minimum of 10% CPU idle time which is not available for processing by FS. Once FS reaches 90% CPU consumption it will respond with cause code 503 to additional SIP requests until its own usage drops below 90%, while reserving that last 10% for other processes on the machine.

pause

```
Usage: fsctl pause [inbound|outbound]
```

Pauses the ability to receive inbound or originate outbound calls, or both directions if the keyword is omitted. Executing fsctl pause inbound will also prevent registration requests from being processed. Executing fsctl pause outbound will result in the Critical log message "The system cannot create any outbound sessions at this time" in the FS log.

Use resume with the corresponding argument to restore normal operation.

pause_check

```
Usage: fsctl pause_check [inbound|outbound]
```

Returns true if the specified mode is active.

Examples:

```
fsctl pause_check inbound
true
```

indicates that inbound calls and registrations are paused. Use fsctl resume inbound to restore normal operation.

```
fsctl pause_check
true
```

indicates that both inbound and outbound sessions are paused. Use fsctl resume to restore normal operation.

ready_check

```
Usage: fsctl ready_check
```

Returns true if the system is in the ready state, as opposed to awaiting an elegant shutdown or other not-ready state.

reclaim_mem

```
Usage: fsctl reclaim_mem
```

recover

```
Usage: fsctl recover
```

Sends an endpoint–specific recover command to each channel detected as recoverable. This replaces “sofia recover” and makes it possible to have multiple endpoints besides SIP implement recovery.

resume

```
Usage: fsctl resume [inbound|outbound]
```

Resumes normal operation after pausing inbound, outbound, or both directions of call processing by FreeSWITCH.

Example:

```
fsctl resume inbound
+OK
```

Resumes processing of inbound calls and registrations. Note that this command always returns +OK, but the same keyword must be used that corresponds to the one used in the pause command in order to take effect.

save_history

```
Usage: fsctl save_history
```

Write out the command history in anticipation of executing a configuration that might crash FS. This is useful when debugging a new module or script to allow other developers to see what commands were executed before the crash.

send_sighup

```
Usage: fsctl send_sighup
```

Does the same thing that killing the FS process with -HUP would do without having to use the UNIX kill command. Useful in environments like Windows where there is no kill command or in cron or other scripts by using fs_cli -x "fsctl send_sighup" where the FS user process might not have privileges to use the UNIX kill command.

shutdown

```
Usage: fsctl shutdown [asap|asap restart|cancel|elegant|now|restart|restart asap|restart elegant]
```

- cancel - discontinue a previous shutdown request.
- elegant - wait for all traffic to stop, while allowing new traffic.
- asap - wait for all traffic to stop, but deny new traffic.
- now - shutdown FreeSWITCH immediately.
- restart - restart FreeSWITCH immediately following the shutdown.

When giving "elegant", "asap" or "now" it's also possible to add the restart command:

shutdown_check

```
Usage: fsctl shutdown_check
```

Returns true if FS is shutting down, or shutting down and restarting.

sps

```
Usage: fsctl sps [int]
```

This changes the sessions-per-second limit from the value initially set in switch.conf

sync_clock

```
Usage: fsctl sync_clock
```

FreeSWITCH will not trust the system time. It gets one sample of system time when it first starts and uses the monotonic clock after that moment. You can sync it back to the current value of the system's real-time clock with fsctl sync_clock

Note: fsctl sync_clock immediately takes effect, which can affect the times on your CDRs. You can end up underbilling/overbilling, or even calls hungup before they originated. e.g. if FS clock is off by 1 month, then your CDRs will show calls that lasted for 1 month!

See fsctl sync_clock_when_idle which is much safer.

sync_clock_when_idle

```
Usage: fsctl sync_clock_when_idle
```

Synchronize the FreeSWITCH clock to the host machine's real-time clock, but wait until there are 0 channels in use. That way it doesn't affect any CDRs.

verbose_events

```
Usage: fsctl verbose_events [on|off]
```

Enables verbose events. Verbose events have **every** channel variable in **every** event for a particular channel. Non-verbose events have only the pre-selected channel variables in the event headers.

See switch.conf.xml for the persistent setting of verbose-channel-events.

global_getvar

Gets the value of a global variable. If the parameter is not provided then it gets all the global variables.

```
Usage: global_getvar [<varname>]
```

global_setvar

Sets the value of a global variable.

```
Usage: global_setvar <varname>=<value>
```

Example:

```
global_setvar outbound_caller_id=2024561000
```

group_call

Returns the bridge string defined in a call group.

```
Usage: group_call group@domain[+F|+A|+E]
```

+F will return the group members in a serial fashion separated by | (the pipe character)

+A (default) will return them in a parallel fashion separated by , (comma)

+E will return them in a enterprise fashion separated by :_: (colon underscore colon).

There is no space between the domain and the optional flag. See Groups in the XML User Directory for more information.

Please note: If you need to have outgoing user variables set in leg B, make sure you don't have dial-string and group-dial-string in your domain or dialed group variables list; instead set dial-string or group-dial-string in the default group of the user. This way group_call will return user/101 and user/ would set all your user variables to the leg B channel.

The B leg receives a new variable, `dialed_group`, containing the full group name.

help

Show help for all the API commands.

```
Usage: help
```

host_lookup

Performs a DNS lookup on a host name.

```
Usage: host_lookup <hostname>
```

hupall

Disconnect existing channels.

```
Usage: hupall <cause> [<variable> <value>]
```

All channels with <variable> set to <value> will be disconnected with <cause> code.

Example:

```
originate {foo=bar}sofia/internal/someone1@server.com,sofia/internal/someone2@server.com &park
```

```
hupall normal_clearing foo bar
```

To hang up all calls on the switch indiscriminately:

```
hupall system_shutdown
```

in_group

Determine if a user is a member of a group.

```
Usage: in_group <user>[@<domain>] <group_name>
```

is_lan_addr

See if an IP is a LAN address.

```
Usage: is_lan_addr <ip>
```

json

JSON API

```
Usage: json {"command" : "...", "data" : "..."}
```

Example

```
> json {"command" : "status", "data" : ""}

{"command":"status","data":"","status":"success","response":{"systemStatus":"ready","uptime":{"years":0,"days":
20,"hours":20,"minutes":37,"seconds":4,"milliseconds":254,"microseconds":44},"version":"1.6.9 -16-d574870
64bit","sessions":{"count":{"total":132,"active":0,"peak":2,"peak5Min":0,"limit":1000},"rate":{"current":0,"
max":30,"peak":2,"peak5Min":0}},"idleCPU":{"used":0,"allowed":99.733333},"stackSizeKB":{"current":240,"max":
8192}}}
```

load

Load external module

```
Usage: load <mod_name>
```

Example:

```
load mod_v8
```

md5

Return MD5 hash for the given input data

```
Usage: md5 hash-key
```

Example:

```
md5 freeswitch-is-awesome
765715d4f914bf8590d1142b6f64342e
```

module_exists

Check if module is loaded.

```
Usage: module_exists <module>
```

Example:

```
module_exists mod_event_socket
true
```

msleep

Sleep for x number of milliseconds

```
Usage: msleep <number of milliseconds to sleep>
```

nat_map

Manage Network Address Translation mapping.

```
Usage: nat_map [status|reinit|republish] | [add|del] <port> [tcp|udp] [sticky] | [mapping] <enable|disable>
```

- status - Gives the NAT type, the external IP, and the currently mapped ports.
- reinit - Completely re-initializes the NAT engine. Use this if you have changed routes or have changed your home router from NAT mode to UPnP mode.
- republish - Causes FreeSWITCH to republish the NAT maps. This should not be necessary in normal operation.
- mapping - Controls whether port mapping requests will be sent to the NAT (the command line option of -nonatmap can set it to disable on startup). This gives the ability of still using NAT for getting the public IP without opening the ports in the NAT.

Note: sticky makes the mapping stay across FreeSWITCH restarts. It gives you a permanent mapping.

Warning: If you have multiple network interfaces with unique IP addresses defined in sip profiles using the same port, nat_map *will* get confused when it tries to map the same ports for multiple profiles. Set up a static mapping between the public address and port and the private address and port in the sip_profiles to avoid this problem.

regex

Evaluate a regex (regular expression).

```
Usage: regex <data>|<pattern>[|<subst string>][|(n|b)]
       regex m:/<data>/<pattern>[/<subst string>][/(n|b)]
       regex m:~<data>~<pattern>[~<subst string>][~(n|b)]
```

This command behaves differently depending upon whether or not a substitution string and optional flag is supplied:

- If a subst is not supplied, regex returns either "true" if the pattern finds a match or "false" if not.
- If a subst is supplied, regex returns the subst value on a true condition.
- If a subst is supplied, on a false (no pattern match) condition regex returns:
    - the source string with no flag;
    - with the n flag regex returns null which forces the response "-ERR no reply" from regex;
    - with the b flag regex returns "false"

The regex delimiter defaults to the | (pipe) character. The delimiter may be changed to ~ (tilde) or / (forward slash) by prefixing the regex with m:

Examples:

```
regex test1234|\d                  <== Returns "true"
regex m:/test1234/\d               <== Returns "true"
regex m:~test1234~\d               <== Returns "true"
regex test|\d                      <== Returns "false"
regex test1234|(\d+)|$1            <== Returns "1234"
regex sip:foo@bar.baz|^sip:(.*)|$1 <== Returns "foo@bar.baz"
regex testingonetwo|(\d+)|$1       <== Returns "testingonetwo" (no match)
regex m:~30~/^(10|20|40)$/~$1      <== Returns "30" (no match)
regex m:~30~/^(10|20|40)$/~$1~n    <== Returns "-ERR no reply" (no match)
regex m:~30~/^(10|20|40)$/~$1~b    <== Returns "false" (no match)
```

Logic in revision 14727 if the source string matches the result then the condition was false however there was a match and it is 1001.

```
regex 1001|/(^\d{4}$)/|$1
```

- See also Regular_Expression

reload

Reload a module.

```
Usage: reload <mod_name>
```

reloadacl

Reload Access Control Lists after modifying them in autoload_configs/acl.conf.xml and as defined in extensions in the user directory conf/directory/*.xml

```
Usage: reloadacl [reloadxml]
```

reloadxml

Reload conf/freeswitch.xml settings after modifying configuration files.

```
Usage: reloadxml
```

show

Display various reports, **VERY** useful for troubleshooting and confirming proper configuration of FreeSWITCH. Arguments can not be abbreviated, they must be specified fully.

```
Usage: show [
   aliases |
   api |
   application |
   bridged_calls |
   calls [count] |
   channels [count|like <match string>] |
   chat |
   codec |
   complete |
   detailed_bridged_calls |
   detailed_calls |
   dialplan |
   endpoint |
   file |
   interface_types |
   interfaces |
   limits
   management |
   modules |
   nat_map |
   registrations |
   say |
   tasks |
   timer |
   ] [as xml|as delim <delimiter>]
```

XML formatted:

```
show foo as xml
```

JSON formatted:

show foo as json

Example

```
fs_cli -x "show channels as json" | jq
```

Example output


```
{
  "row_count": 1,
  "rows": [
    {
      "uuid": "aa47bc9c-ab5e-11ea-85f1-311ce82e049e",
      "direction": "inbound",
      "created": "2020-06-10 17:09:26",
      "created_epoch": "1591823366",
      "name": "sofia/internal/1019@192.0.2.10",
      "state": "CS_EXECUTE",
      "cid_name": "1019",
      "cid_num": "1019",
      "ip_addr": "192.0.2.50",
      "dest": "55",
      "application": "echo",
      "application_data": "",
      "dialplan": "XML",
      "context": "default",
      "read_codec": "PCMU",
      "read_rate": "8000",
      "read_bit_rate": "64000",
      "write_codec": "PCMU",
      "write_rate": "8000",
      "write_bit_rate": "64000",
      "secure": "",
      "hostname": "hostname.local",
      "presence_id": "1019@192.0.2.10",
      "presence_data": "",
      "accountcode": "1019",
      "callstate": "ACTIVE",
      "callee_name": "",
      "callee_num": "",
      "callee_direction": "",
      "call_uuid": "",
      "sent_callee_name": "",
      "sent_callee_num": "",
      "initial_cid_name": "1019",
      "initial_cid_num": "1019",
      "initial_ip_addr": "192.0.2.50",
      "initial_dest": "55",
      "initial_dialplan": "XML",
      "initial_context": "default"
    }
  ]
}
```


Change delimiter:

```
show foo as delim |
```

- aliases – list defined command aliases
- api – list API commands exposed by loadable modules
- application – list applications exposed by loadable modules, notably mod_dptools

- bridged_calls – deprecated, use "show calls"
- calls [count] – list details of currently active calls; the keyword "count" eliminates the details and only prints the total count of calls
- channels [count|like <match string>] – list current channels; see Channels vs Calls
    - count – show only the count of active channels, no details
    - like <match string> – filter results to include only channels that contain <match string> in uuid, channel name, cid_number, cid_name, presence data fields.
- chat – list chat interfaces
- codec – list codecs that are currently loaded in FreeSWITCH
- complete – list command argument completion tables
- detailed_bridged_calls – same as "show detailed_calls"
- detailed_calls – like "show calls" but with more fields
- dialplan – list dialplan interfaces
- endpoint – list endpoint interfaces currently available to FS
- file – list supported file format interfaces
- interface_types – list all interface types with a summary count of each type of interface available
- interfaces – enumerate all available interfaces by type, showing the module which exposes each interface
- limits – list database limit interfaces
- management – list management interfaces
- module – enumerate modules and the path to each
- nat_map – list Network Address Translation map
- registrations – enumerate user extension registrations
- say – enumerate available TTS (text-to-speech) interface modules with language supported
- tasks – list FS tasks
- timer – list timer modules

Tips For Showing Calls and Channels

The best way to get an understanding of all of the show calls/channels is to use them and observe the results. To display more fields:

- show detailed_calls
- show bridged_calls
- show detailed_bridged_calls

These three take the expand on information shown by "show calls". Note that "show detailed_calls" replaces "show distinct_channels". It provides similar, but more detailed, information. Also note that there is no "show detailed_channels" command, however using "show detailed_calls" will yield the same net result: FreeSWITCH lists detailed information about one-legged calls and bridged calls by using "show detailed_calls", which can be quite useful while configuring and troubleshooting FS.

Filtering Results

To filter only channels matching a specific uuid or related to a specific call, set the presence_data channel variable in the bridge or originate application to a unique string. Then you can use:

```
show channels like foo
```

to list only those channels of interest. The **like** directive filters on these fields:

- uuid
- channel name
- caller id name
- caller id number
- presence_data

NOTE: **presence_data** must be set during **bridge** or **originate** and not after the channel is established.

shutdown

Stop the FreeSWITCH program.

```
Usage: shutdown
```

This only works from the console. To shutdown FS from an API call or fs_cli, you should use "fsctl shutdown" which offers a number of options.

Shutdown from the console ignores arguments and exits immediately!

status

Show current FS status. Very helpful information to provide when asking questions on the mailing list or irc channel.

```
Usage: status
```

```
freeswitch@internal> status
UP 17 years, 20 days, 10 hours, 10 minutes, 31 seconds, 571 milliseconds, 721 microseconds
FreeSWITCH (Version 1.5.8b git 87751f9 2013-12-13 18:13:56Z 32bit) is ready  <!-- FS version -->
53987253 session(s) since startup                                            <!-- cumulative total number of
channels created since FS started -->
127 session(s) - peak 127, last 5min 253                                     <!-- current number of active
channels -->
55 session(s) per Sec out of max 60, peak 55, last 5min 253                  <!-- current channels per second
created, max cps set in switch.conf.xml -->
1000 session(s) max                                                          <!-- set in switch.conf.xml -->
min idle cpu 0.00/97.71                                                      <!-- minimum reserved idle CPU time
before refusing new calls, set in switch.conf.xml -->
```

strftime_tz

Displays formatted time, converted to a specific timezone. See /usr/share/zoneinfo/zone.tab for the standard list of Linux timezones.

```
Usage: strftime_tz <timezone> [format_string]
```

Example:

```
strftime_tz US/Eastern %Y-%m-%d %T
```

unload

Unload external module.

```
Usage: unload <mod_name>
```

version

Show version of the switch

```
Usage: version [short]
```

Examples:

```
freeswitch@internal> version
FreeSWITCH Version 1.5.8b+git~20131213T181356Z~87751f9eaf~32bit (git 87751f9 2013-12-13 18:13:56Z 32bit)

freeswitch@internal> version short
1.5.8b
```

xml_locate

Write active xml tree or specified branch to stdout.

```
Usage: xml_locate [root | <section> | <section> <tag> <tag_attr_name> <tag_attr_val>]
```

xml_locate root will return all XML being used by FreeSWITCH

xml_locate <section>: Will return the XML corresponding to the specified <section>

```
xml_locate directory
xml_locate configuration
xml_locate dialplan
xml_locate phrases
```

Example:

```
xml_locate directory domain name example.com

xml_locate configuration configuration name ivr.conf
```

xml_wrap

Wrap another API command in XML.

```
Usage: xml_wrap <command> <args>
```

Call Management Commands

break

Deprecated. See uuid_break.

create_uuid

Creates a new UUID and returns it as a string.

```
Usage: create_uuid
```

originate

Originate a new call.

Usage

```
originate <call_url> <exten>|&<application_name>(<app_args>) [<dialplan>] [<context>] [<cid_name>] [<cid_num>]
[<timeout_sec>]
```

FreeSWITCH will originate a call to <call_url> as Leg A. If that leg supervises within 60 seconds FS will continue by searching for an extension definition in the specified dialplan for <exten> or else execute the application that follows the & along with its arguments.

Originate Arguments

Arguments

- <call_url> URL you are calling. For more info on sofia SIP URL syntax see: FreeSwitch Endpoint Sofia
- Destination, one of:
    - <exten> Destination number to search in dialplan; note that registered extensions will fail this way, use &bridge(user/xxxx) instead
    - &<application_name>(<app_args>)
        - "&" indicates what follows is an application name, not an exten
        - (<app_args>) is optional (not all applications require parameters, e.g. park)
        - The most commonly used application names include:
          park, bridge, javascript/lua/perl, playback (remove mod_native_file).
        - Note: Use single quotes to pass arguments with spaces, e.g. '&lua(test.lua arg1 arg2)'
        - Note: There is no space between & and the application name
- <dialplan> Defaults to 'XML' if not specified.
- <context> Defaults to 'default' if not specified.
- <cid_name> CallerID name to send to Leg A.
- <cid_num> CallerID number to send to Leg A.
- <timeout_sec> Timeout in seconds; default = 60 seconds.

Originate Variables

Variables

These variables can be prepended to the dial string inside curly braces and separated by commas. Example:

```
originate {sip_auto_answer=true,return_ring_ready=false}user/1001 9198
```

Variables within braces must be separated by a comma.

- group_confirm_key
- group_confirm_file
- forked_dial
- fail_on_single_reject
- ignore_early_media - must be defined on Leg B in bridge or originate command to stop remote ringback from being heard by Leg A
- return_ring_ready
- originate_retries
- originate_retry_sleep_ms
- origination_caller_id_name
- origination_caller_id_number
- originate_timeout
- sip_auto_answer

Description of originate's related variables

Originate Examples

Examples

You can call a locally registered sip endpoint 300 and park the call like so Note that the "example" profile used here must be the one to which 300 is registered. Also note the use of % instead of @ to indicate that it is a registered extension.

```
originate sofia/example/300%pbx.internal &park()
```

Or you could instead connect a remote sip endpoint to extension 8600

```
originate sofia/example/300@foo.com 8600
```

Or you could instead connect a remote SIP endpoint to another remote extension

```
originate sofia/example/300@foo.com &bridge(sofia/example/400@bar.com)
```

Or you could even run a Javascript application test.js

```
originate sofia/example/1000@somewhere.com &javascript(test.js)
```

To run a javascript with arguments you must surround it in single quotes.

```
originate sofia/example/1000@somewhere.com '&javascript(test.js myArg1 myArg2)'
```

Setting channel variables to the dial string

```
originate {ignore_early_media=true}sofia/mydomain.com/18005551212@1.2.3.4 15555551212
```

Setting SIP header variables to send to another FS box during originate

```
originate {sip_h_X-varA=111,sip_h_X-varB=222}sofia/mydomain.com/18005551212@1.2.3.4 15555551212
```

Note: you can set any channel variable, even custom ones. Use single quotes to enclose values with spaces, commas, etc.

```
originate {my_own_var=my_value}sofia/mydomain.com/that.ext@1.2.3.4 15555551212
originate {my_own_var='my value'}sofia/mydomain.com/that.ext@1.2.3.4 15555551212
```

If you need to fake the ringback to the originated endpoint try this:

```
originate {ringback=\'%(2000,4000,440.0,480.0)\'}sofia/example/300@foo.com &bridge(sofia/example/400@bar.com)
```

To specify a parameter to the Leg A call and the Leg B bridge application:

```
originate {'origination_caller_id_number=2024561000'}sofia/gateway/whitehouse.gov/2125551212 &bridge
(['effective_caller_id_number=7036971379']sofia/gateway/pentagon.gov/3035554499)
```

If you need to make originate return immediately when the channel is in "Ring-Ready" state try this:

```
   originate {return_ring_ready=true}sofia/gateway/someprovider/919246461929 &socket('127.0.0.1:8082 async
full')
```

More info on return_ring_ready

You can even set music on hold for the ringback if you want:

```
   originate {ringback=\'/path/to/music.wav\'}sofia/gateway/name/number &bridge(sofia/gateway/siptoshore
/12425553741)
```

You can originate a call in the background (asynchronously) and playback a message with a 60 second timeout.

```
   bgapi originate {ignore_early_media=true,originate_timeout=60}sofia/gateway/name/number &playback(message)
```

You can specify the UUID of an originated call by doing the following:

- Use create_uuid to generate a UUID to use.
- This will allow you to kill an originated call before it is answered by using uuid_kill.
- If you specify `origination_uuid`, it will remain the UUID for the answered call leg for the whole session.

```
originate {origination_uuid=...}user/100@domain.name.com
```

TODO Event List#1.21CHANNEL_UUIDevent also mention the `origination_uuid` in conjunction with the `bridge` command, but couldn't find it documented there.

Here's an example of originating a call to the echo conference (an external sip URL) and bridging it to a local user's phone:

```
   originate sofia/internal/9996@conference.freeswitch.org &bridge(user/105@default)
```

Here's an example of originating a call to an extension in a different context than 'default' (required for the FreePBX which uses context_1, context_2, etc.):

```
   originate sofia/internal/2001@foo.com 3001 xml context_3
```

You can also originate to multiple extensions as follows:

```
   originate user/1001,user/1002,user/1003 &park()
```

To put an outbound call into a conference at early media, either of these will work (they are effectively the same thing)

```
   originate sofia/example/300@foo.com &conference(conf_uuid-TEST_CON)
   originate sofia/example/300@foo.com conference:conf_uuid-TEST_CON inline
```

See mod_dptools: Inline Dialplan for more detail on 'inline' Dialplans

An example of using loopback and inline on the A-leg can be found in this mailing list post

pause

Pause <uuid> playback of recorded media that was started with uuid_broadcast.

Usage
`pause <uuid> <on|off>`

Turning pause "on" activates the pause function, i.e. it pauses the playback of recorded media. Turning pause "off" deactivates the pause function and resumes playback of recorded media at the same point where it was paused.

Note: always returns -ERR no reply when successful; returns -ERR No such channel! when uuid is invalid.

uuid_answer

Answer a channel

Usage
`uuid_answer <uuid>`

See Also

- mod_dptools: answer

uuid_audio

Adjust the audio levels on a channel or mute (read/write) via a media bug.

Usage
`uuid_audio <uuid> [start [read|write] [[mute|level] <level>]|stop]`

<level> is in the range from -4 to 4, 0 being the default value.

Level is required for both mute|level params:

```
freeswitch@internal> uuid_audio 0d7c3b93-a5ae-4964-9e4d-902bba50bd19 start write mute <level>
freeswitch@internal> uuid_audio 0d7c3b93-a5ae-4964-9e4d-902bba50bd19 start write level <level>
```

(This command behaves funky. Requires further testing to vet all arguments. - JB)

See Also

- mod_dptools: set audio level

uuid_break

Break out of media being sent to a channel. For example, if an audio file is being played to a channel, issuing uuid_break will discontinue the media and the call will move on in the dialplan, script, or whatever is controlling the call.

`Usage: uuid_break <uuid> [all]`

If the **all** flag is used then all audio files/prompts/etc. that are queued up to be played to the channel will be stopped and removed from the queue, otherwise only the currently playing media will be stopped.

uuid_bridge

Bridge two call legs together.

Usage
`uuid_bridge <uuid> <other_uuid>`

uuid_bridge needs at least any one leg to be in the answered state. If, for example, one channel is parked and another channel is actively conversing on a call, executing uuid_bridge on these 2 channels will drop the existing call and bridge together the specified channels.

> **mod_dptools: bridge VS uuid_bridge**
>
> https://lists.freeswitch.org/pipermail/freeswitch-users/2014-September/108166.html

uuid_broadcast

Execute an arbitrary dialplan application, typically playing a media file, on a specific uuid. If a filename is specified then it is played into the channel(s). To execute an application use "app::args" syntax.

Usage
`uuid_broadcast <uuid> <path> [aleg|bleg|both]`

Execute an application on a chosen leg(s) with optional hangup afterwards:

Usage

```
uuid_broadcast <uuid> app[![hangup_cause]]::args [aleg|bleg|both]
```

Here are some examples:

Examples

```
uuid_broadcast 336889f2-1868-11de-81a9-3f4acc8e505e sorry.wav both
uuid_broadcast 336889f2-1868-11de-81a9-3f4acc8e505e say::en\snumber\spronounced\s12345 aleg
uuid_broadcast 336889f2-1868-11de-81a9-3f4acc8e505e say!::en\snumber\spronounced\s12345 aleg
uuid_broadcast 336889f2-1868-11de-81a9-3f4acc8e505e say!user_busy::en\snumber\spronounced\s12345 aleg
uuid_broadcast 336889f2-1868-11de-81a9-3f4acc8e505e playback!user_busy::sorry.wav aleg
```

uuid_buglist

List the media bugs on channel. Output is formatted as XML.

Usage

```
uuid_buglist <uuid>
```

Example

```
uuid_buglist c2746178-ab61-11ea-86b8-311ce82e049e
```

Example output

```
<media-bugs>
 <media-bug>
  <function>session_record</function>
  <target>rtmp://domain.local/stream:teststream</target>
  <thread-locked>0</thread-locked>
 </media-bug>
</media-bugs>
```

uuid_chat

Send a chat message.

Usage

```
uuid_chat <uuid> <text>
```

If the endpoint associated with the session <uuid> has a receive_event handler, this message gets sent to that session and is interpreted as an instant message.

uuid_debug_media

Debug media, either audio or video.

Usage

```
uuid_debug_media <uuid> <read|write|both|vread|vwrite|vboth> <on|off>
```

Use "read" or "write" for the audio direction to debug, or "both" for both directions. And prefix with v for video media.

uuid_debug_media emits a HUGE amount of data. If you invoke this command from fs_cli, be prepared.

Example output

```
R sofia/internal/1003@192.168.65.3 b= 172 192.168.65.3:17668 192.168.65.114:16072 192.168.65.114:16072 pt=0
ts=2981605109 m=0
W sofia/internal/1003@192.168.65.3 b= 172 192.168.65.3:17668 192.168.65.114:16072 192.168.65.114:16072 pt=0
ts=12212960 m=0
R sofia/internal/1003@192.168.65.3 b= 172 192.168.65.3:17668 192.168.65.114:16072 192.168.65.114:16072 pt=0
ts=2981605269 m=0
W sofia/internal/1003@192.168.65.3 b= 172 192.168.65.3:17668 192.168.65.114:16072 192.168.65.114:16072 pt=0
ts=12213120 m=0
```

Read Format

"R %s b=%4ld %s:%u %s:%u pt=%d ts=%u m=%d\n"

where the values are:

- switch_channel_get_name(switch_core_session_get_channel(session)),
- (long) bytes,
- my_host, switch_sockaddr_get_port(rtp_session->local_addr),
- old_host, rtp_session->remote_port,
- tx_host, switch_sockaddr_get_port(rtp_session->from_addr),
- rtp_session->recv_msg.header.pt,
- ntohl(rtp_session->recv_msg.header.ts),
- rtp_session->recv_msg.header.m

Write Format

"W %s b=%4ld %s:%u %s:%u pt=%d ts=%u m=%d\n"

where the values are:

- switch_channel_get_name(switch_core_session_get_channel(session)),
- (long) bytes,
- my_host, switch_sockaddr_get_port(rtp_session->local_addr),
- old_host, rtp_session->remote_port,
- tx_host, switch_sockaddr_get_port(rtp_session->from_addr),
- send_msg->header.pt,
- ntohl(send_msg->header.ts),
- send_msg->header.m);

uuid_deflect

Deflect an answered SIP call off of FreeSWITCH by sending the REFER method

```
Usage: uuid_deflect <uuid> <sip URL>
```

uuid_deflect waits for the final response from the far end to be reported. It returns the sip fragment from that response as the text in the FreeSWITCH response to uuid_deflect. If the far end reports the REFER was successful, then FreeSWITCH will issue a bye on the channel.

Example

```
uuid_deflect 0c9520c4-58e7-40c4-b7e3-819d72a98614 sip:info@example.net
```

Response:

```
  Content-Type: api/response
  Content-Length: 30

  +OK:SIP/2.0 486 Busy Here
```

uuid_displace

Displace the audio for the target <uuid> with the specified audio <file>.

```
Usage: uuid_displace <uuid> [start|stop] <file> [<limit>] [mux]
```

Arguments:

- uuid = Unique ID of this call (see 'show channels')
- start|stop = Start or stop this action
- file = path to an audio source (.wav file, shoutcast stream, etc...)
- limit = limit number of seconds before terminating the displacement
- mux = multiplex; mix the original audio together with 'file', i.e. both parties can still converse while the file is playing (if the level is not too loud)

To specify the 5th argument 'mux' you must specify a limit; if no time limit is desired on playback, then specify 0.

Examples

```
cli> uuid_displace 1a152be6-2359-11dc-8f1e-4d36f239dfb5 start /sounds/test.wav 60
cli> uuid_displace 1a152be6-2359-11dc-8f1e-4d36f239dfb5 stop /sounds/test.wav
```

uuid_display

Updates the display on a phone if the phone supports this. This works on some SIP phones right now including Polycom and Snom.

```
Usage: <uuid> name|number
```

Note the pipe character separating the Caller ID name and Caller ID number.

This command makes the phone re-negotiate the codec. The SIP -> RTP Packet Size should be 0.020 seconds. If it is set to 0.030 on the Cisco SPA series phones it causes a DTMF lag. When DTMF keys are pressed on the phone they are can be seen on the fs_cli 4-6 seconds late.

Example:

```
freeswitch@sidious> uuid_display f4053af7-a3b9-4c78-93e1-74e529658573 Fred Jones|1001

+OK Success
```

uuid_dual_transfer

Transfer each leg of a call to different destinations.

```
Usage: <uuid> <A-dest-exten>[/<A-dialplan>][/<A-context>] <B-dest-exten>[/<B-dialplan>][/<B-context>]
```

uuid_dump

Dumps all variable values for a session.

```
Usage: uuid_dump <uuid> [format]
```

Format options: txt (default, may be omitted), XML, JSON, plain

uuid_early_ok

Stops the process of ignoring early media, i.e. if ignore_early_media=true, this stops ignoring early media coming from Leg B and responds normally.

```
Usage: uuid_early_ok <uuid>
```

uuid_exists

Checks whether a given UUID exists.

```
Usage: uuid_exists <uuid>
```

Returns true or false.

uuid_flush_dtmf

Flush queued DTMF digits

```
Usage: uuid_flush_dtmf <uuid>
```

uuid_fileman

Manage the audio being played into a channel from a sound file

```
Usage: uuid_fileman <uuid> <cmd:val>
```

Commands are:

- speed:<+[step]>|<-[step]>
- volume:<+[step]>|<-[step]>
- pause (toggle)
- stop
- truncate
- restart
- seek:<+[milliseconds]>|<-[milliseconds]> (1000ms = 1 second, 10000ms = 10 seconds.)

Example to seek forward 30 seconds:

```
uuid_fileman 0171ded1-2c31-445a-bb19-c74c659b7d08 seek:+3000
```

(Or use the current channel via ${uuid}, e.g. in a bind_digit_action)

The 'pause' argument is a toggle: the first time it is invoked it will pause playback, the second time it will resume playback.

uuid_getvar

Get a variable from a channel.

```
Usage: uuid_getvar <uuid> <varname>
```

uuid_hold

Place a channel on hold.

Usage:

```
uuid_hold <uuid>           place a call on hold
uuid_hold off <uuid>       switch off on hold
uuid_hold toggle <uuid>    toggles call-state based on current call-state
```

uuid_kill

Reset a specific <uuid> channel.

```
Usage: uuid_kill <uuid> [cause]
```

If no cause code is specified, NORMAL_CLEARING will be used.

uuid_limit

Apply or change limit(s) on a specified uuid.

```
Usage: uuid_limit <uuid> <backend> <realm> <resource> [<max>[/interval]] [number [dialplan [context]]]
```

See also mod_dptools: limit

uuid_media

Reinvite FreeSWITCH out of the media path:

```
Usage: uuid_media [off] <uuid>
```

Reinvite FreeSWITCH back in:

Usage: uuid_media <uuid>

uuid_media_reneg

Tell a channel to send a re-invite with optional list of new codecs to be renegotiated.

```
Usage: uuid_media_reneg <uuid> <=><codec string>
```

Example: Adding =PCMU makes the offered codec string absolute.

uuid_park

Park call

```
Usage: uuid_park <uuid>
```

The specified channel will be parked and the other leg of the call will be disconnected.

uuid_pre_answer

Pre–answer a channel.

```
Usage: uuid_preanswer <uuid>
```

See Also: Misc._Dialplan_Tools_pre_answer

uuid_preprocess

Pre-process Channel

```
Usage: uuid_preprocess <uuid>
```

uuid_recv_dtmf

Usage: uuid_recv_dtmf <uuid> <dtmf_data>

uuid_send_dtmf

Send DTMF digits to <uuid>

```
Usage: uuid_send_dtmf <uuid> <dtmf digits>[@<tone_duration>]
```

Use the character w for a .5 second delay and the character W for a 1 second delay.

Default tone duration is 2000ms .

uuid_send_info

Send info to the endpoint

```
Usage: uuid_send_info <uuid>
```

uuid_session_heartbeat

```
Usage: uuid_session_heartbeat <uuid> [sched] [0|<seconds>]
```

uuid_setvar

Set a variable on a channel. If value is omitted, the variable is unset.

```
Usage: uuid_setvar <uuid> <varname> [value]
```

Dialplan equivalent

```
<action application="set" data="<varname>=<value>"/>

<!-- For example: -->
<action application="set" data="playback_terminators=none"/>
```

Example

```
uuid_setvar c2746178-ab61-11ea-86b8-311ce82e049e record_sample_rate 8000
```

uuid_setvar_multi

Set multiple vars on a channel.

```
Usage: uuid_setvar_multi <uuid> <varname>=<value>[;<varname>=<value>[;...]]
```

uuid_simplify

This command directs FreeSWITCH to remove itself from the SIP signaling path if it can safely do so.

```
Usage: uuid_simplify <uuid>
```

Execute this API command to instruct FreeSWITCH™ to inspect the Leg A and Leg B network addresses. If they are both hosted by the same switch as a result of a transfer or forwarding loop across a number of FreeSWITCH™ systems the one executing this command will remove itself from the SIP and media path and restore the endpoints to their local FreeSWITCH™ to shorten the network path. This is particularly useful in large distributed FreeSWITCH™ installations.

For example, suppose a call arrives at a FreeSWITCH™ box in Los Angeles, is answered, then forwarded to a FreeSWITCH™ box in London, answered there and then forwarded back to Los Angeles. The London switch could execute uuid_simplify to tell its local switch to examine both legs of the call to determine that they could be hosted by the Los Angeles switch since both legs are local to it. Alternatively, setting sip_auto_simplify to true either globally in vars.xml or as part of a dailplan extension would tell FS to perform this check for each call when both legs supervise.

uuid_transfer

TODO What is the difference between `uuid_transfer` in `mod_commands` and `mod_dptools: transfer`?

Transfers an existing call to a specific extension within a <dialplan> and <context>. Dialplan may be "xml" or "directory".

Usage

```
uuid_transfer <uuid> [-bleg|-both] <dest-exten> [<dialplan>] [<context>]
```

The optional first argument will allow you to transfer both parties (-both) or only the party to whom <uuid> is talking.(-bleg). Beware that -bleg actually means "the other leg", so when it is executed on the actual B leg uuid it will transfer the actual A leg that originated the call and disconnect the actual B leg.

NOTE: if the call has been bridged, and you want to transfer either side of the call, then you will need to use <action application="set" data="hangup_after_bridge=false"/> (or the API equivalent). If it's not set, transfer doesn't really work as you'd expect, and leaves calls in limbo.

And more examples see Inline Dialplan

uuid_phone_event

Send hold indication upstream:

Usage
`uuid_phone_event <uuid> hold|talk`

Record/Playback Commands

uuid_record

Record the audio associated with the channel with the given UUID into a file. The start command causes FreeSWITCH to start mixing all call legs together and saves the result as a file in the format that the file's extension dictates. The stop command (if available) will stop the recording and close the file.

This API command is possibly broken and the "*(if available)*" remark may be an allusion to that: `uuid_record` seems to completely hijack control, no FreeSWITCH events are generated (neither on `fs_cli` nor to an event socket app), and the only way to stop it is by hanging up (as it doesn't seem to honour the `playback_terminators` variable).

Use `mod_dptools:record` or `mod_dptools:record_session` instead.

If media setup hasn't yet happened, the file will contain silent audio until media is available. Audio will be recorded for calls that are parked. The recording will continue through the bridged call. If the call is set to return to park after the bridge, the bug will remain on the call, but no audio is recorded until the call is bridged again.

TODO What if media doesn't flow through FreeSWITCH? Will it re-INVITE first? Or do we just not get the audio in that case?

See channel variables related to recording at `mod_dptools:record`. Another module worth looking at is `mod_dptools: record_session`.

Syntax
`uuid_record <uuid> [start|stop|mask|unmask] <path> [<limit>]`

Parameter	Description
`uuid`	UUID of the channel.
`start`	Start recording the audio.
`stop`	Stop recording. `all` may be used for `<path>` to stop all recordings for the channel with the given UUID. `uuid_record <uuid> stop all`
`mask`	Mask with silence part of the recording beginning when the mask argument is executed by this command. See http://jira.freeswitch.org/browse/FS-5269
`unmask`	Stop the masking and continue recording live audio normally. See http://jira.freeswitch.org/browse/FS-5269
`path`	Record to file specified by given path. If only filename is given the it will be saved to channel variable `sound_prefix`, or `base_dir` when `sound_prefix` not set.
`limit`	(optional) The maximum duration of the recording in seconds.

Limit Commands

More information is available at Limit commands

limit_reset

Reset a limit backend.

limit_status

Retrieve status from a limit backend.

limit_usage

Retrieve usage for a given resource.

uuid_limit_release

Manually decrease a resource usage by one.

limit_interval_reset

Reset the interval counter to zero prior to the start of the next interval.

Miscellaneous Commands

bg_system

Execute a system command in the background.

```
Usage: bg_system <command>
```

echo

Echo input back to the console

```
Usage: echo <text to echo>
```

Example:

```
echo This text will appear
This text will appear
```

file_exists

Tests whether filename exists.

```
file_exists filename
```

Examples:

```
freeswitch> file_exists /tmp/real_file
true

freeswitch> file_exists /tmp/missing_file
false
```

Example dialplan usage:

file_exists example

```
<extension name="play-news-announcements">
  <condition expression="${file_exists(${sounds_dir}/news.wav)}" expression="true"/>
    <action application="playback" data="${sounds_dir}/news.wav"/>
    <anti-action application="playback" data="${soufnds_dir}/no-news-is-good-news.wav"/>
  </condition>
</extension>
```

file_exists tests whether FreeSWITCH can see the file, but the file may still be unreadable because of restrictive permissions.

Example start/stop record to rtmp

```
uuid_record c2746178-ab61-11ea-86b8-311ce82e049e start rtmp://domain.com/stream:teststream

uuid_record c2746178-ab61-11ea-86b8-311ce82e049e stop rtmp://domain.com/stream:teststream
```

find_user_xml

Checks to see if a user exists. Matches user tags found in the directory, similar to user_exists, but returns an XML representation of the user as defined in the directory (like the one shown in user_exists).

`Usage: find_user_xml <key> <user> <domain>`

<key> references a key specified in a directory's user tag

<user> represents the value of the key

<domain> is the domain to which the user is assigned.

list_users

Lists Users configured in Directory

Usage:

`list_users [group <group>] [domain <domain>] [user <user>] [context <context>]`

Examples:

```
freeswitch@localhost> list_users group default

userid|context|domain|group|contact|callgroup|effective_caller_id_name|effective_caller_id_number
2000|default|192.168.20.73|default|sofia/internal/sip:2000@192.168.20.219:5060|techsupport|B#-Test 2000|2000
2001|default|192.168.20.73|default|sofia/internal/sip:2001@192.168.20.150:63412;
rinstance=8e2c8b86809acf2a|techsupport|Test 2001|2001
2002|default|192.168.20.73|default|error/user_not_registered|techsupport|Test 2002|2002
2003|default|192.168.20.73|default|sofia/internal/sip:2003@192.168.20.149:5060|techsupport|Test 2003|2003
2004|default|192.168.20.73|default|error/user_not_registered|techsupport|Test 2004|2004

+OK
```

Search filters can be combined:

```
freeswitch@localhost> list_users group default user 2004

userid|context|domain|group|contact|callgroup|effective_caller_id_name|effective_caller_id_number
2004|default|192.168.20.73|default|error/user_not_registered|techsupport|Test 2004|2004

+OK
```

sched_api

Schedule an API call in the future.

Usage

```
sched_api [+@]<time> <group_name> <command_string>[&]
```

```
<time> is the UNIX timestamp at which the command should be executed. If it is prefixed by +, <time> specifies
the number of seconds to wait before executing the command. If prefixed by @, it will execute the command
periodically every <time> seconds; for the first instance it will be executed after <time> seconds.
```

<group_name> will be the value of "Task-Group" in generated events. "none" is the proper value for no group. If set to UUID of channel (example: ${uuid}), task will automatically be unscheduled when channel hangs up.

<command_string> is the command to execute at the scheduled time.

A scheduled task or group of tasks can be revoked with sched_del or unsched_api.

You could append the "&" symbol to the end of the line to executed this command in its own thread.

Examples

```
sched_api +1800 none originate sofia/internal/1000%${sip_profile} &echo()
sched_api @600 check_sched log Periodic task is running...
sched_api +10 ${uuid} chat verto|fs@mydomain.com|1000@mydomain.com|Hello World
```

sched_broadcast

Play a <path> file to a specific <uuid> call in the future.

Usage

```
sched_broadcast [[+]<time>|@time] <uuid> <path> [aleg|bleg|both]
```

```
Schedule execution of an application on a chosen leg(s) with optional hangup:
```

```
sched_broadcast [+]<time> <uuid> app[![hangup_cause]]::args [aleg|bleg|both]
```

<time> is the UNIX timestamp at which the command should be executed. If it is prefixed by +, <time> specifies the number of seconds to wait before executing the command. If prefixed by @, it will execute the command periodically every <time> seconds; for the first instance it will be executed after <time> seconds.

Examples

```
sched_broadcast +60 336889f2-1868-11de-81a9-3f4acc8e505e commercial.wav both
sched_broadcast +60 336889f2-1868-11de-81a9-3f4acc8e505e say::en\snumber\spronounced\s12345 aleg
```

sched_del

Removes a prior scheduled group or task ID

Usage

```
sched_del <group_name|task_id>
```

```
The one argument can either be a group of prior scheduled tasks or the returned task-id from sched_api.
```

`sched_transfer`, `sched_hangup` and `sched_broadcast` commands add new tasks with group names equal to the channel UUID. Thus, `sched_del` with the channel UUID as the argument will remove all previously scheduled hangups, transfers and broadcasts for this channel.

Examples

```
sched_del my_group
sched_del 2
```

sched_hangup

Schedule a running call to hangup.

Usage

```
sched_hangup [+]<time> <uuid> [<cause>]
```

sched_hangup +0 is the same as uuid_kill

sched_transfer

Schedule a transfer for a running call.

Usage

```
sched_transfer [+]<time> <uuid> <target extension> [<dialplan>] [<context>]
```

stun

Executes a STUN lookup.

Usage:

```
stun <stunserver>[:port]
```

Example:

```
stun stun.freeswitch.org
```

system

Execute a system command.

Usage:

```
system <command>
```

The <command> is passed to the system shell, where it may be expanded or interpreted in ways you don't expect. This can lead to security bugs if you're not careful. For example, the following command is dangerous:

```
<action application="system" data="log_caller_name ${caller_id_name}" />
```

If a malicious remote caller somehow sets his caller ID name to "`; rm -rf /`" you would unintentionally be executing this shell command:

```
log_caller_name; rm -rf /
```

This would be a Bad Thing.

time_test

Runs a test to see how bad timer jitter is. It runs the test <count> times if specified, otherwise it uses the default count of 10, and tries to sleep for mss microseconds. It returns the actual timer duration along with an average.

Usage:

time_test <mss> [count]

Example:

```
time_test 100 5

test 1 sleep 100 99
test 2 sleep 100 97
test 3 sleep 100 96
test 4 sleep 100 97
test 5 sleep 100 102
avg 98
```

timer_test

Runs a test to see how bad timer jitter is. Unlike time_test, this uses the actual FreeSWITCH timer infrastructure to do the timer test and exercises the timers used for call processing.

Usage:

```
timer_test <10|20|40|60|120> [<1..200>] [<timer_name>]
```

The first argument is the timer interval.

The second is the number of test iterations.

The third is the timer name; "show timers" will give you a list.

Example:

```
timer_test 20 3

Avg: 16.408ms Total Time: 49.269ms

2010-01-29 12:01:15.504280 [CONSOLE] mod_commands.c:310 Timer Test: 1 sleep 20 9254
2010-01-29 12:01:15.524351 [CONSOLE] mod_commands.c:310 Timer Test: 2 sleep 20 20042
2010-01-29 12:01:15.544336 [CONSOLE] mod_commands.c:310 Timer Test: 3 sleep 20 19928
```

tone_detect

Start Tone Detection on a channel.

Usage:

tone_detect <uuid> <key> <tone_spec> [<flags> <timeout> <app> <args>] <hits>

<uuid> is required when this is executed as an api call; as a dialplan app the uuid is implicit as part of the channel variables

<key> is an arbitrary name that identifies this tone_detect instance; required

<tone_spec> frequencies to detect; required

<flags> 'r' or 'w' to specify which direction to monitor

<timeout> duration during which to detect tones;

- 0 = detect forever
- +time = number of milliseconds after tone_detect is executed
- time = absolute time to stop in seconds since The Epoch (1 January, 1970)

<app> FS application to execute when tone_detect is triggered; if app is omitted, only an event will be returned

<args> arguments to application enclosed in single quotes

<hits> the number of times tone_detect should be triggered before executing the specified app

Once tone_detect returns a result, it will not trigger again until reset. Reset tone_detect by calling `tone_detect <key>` with no additional arguments to reactivate the previously specified tone_detect declaration.

See also http://wiki.freeswitch.org/wiki/Misc._Dialplan_Tools_tone_detect

unsched_api

Unschedule a previously scheduled API command.

Usage
`unsched_api <task_id>`

url_decode

Usage:

```
url_decode <string>
```

url_encode

Url encode a string.

Usage:

```
url_encode <string>
```

user_data

Retrieves user information (parameters or variables) as defined in the FreeSWITCH user directory.

Usage:

```
user_data <user>@<domain> <attr|var|param> <name>
```

<user> is the user's id

<domain> is the user's domain

<attr|var|param> specifies whether the requested data is contained in the "variables" or "parameters" section of the user's record

<name> is the name (key) of the variable to retrieve

Examples:

```
user_data 1000@192.168.1.101 param password
```

will return a result of 1234, and

```
user_data 1000@192.168.1.101 var accountcode
```

will return a result of 1000 from the example user shown in user_exists, and

```
user_data 1000@192.168.1.101 attr id
```

will return the user's actual alphanumeric ID (i.e. "john") when number-alias="1000" was set as an attribute for that user.

user_exists

Checks to see if a user exists. Matches user tags found in the directory and returns either true/false:

Usage:

```
user_exists <key> <user> <domain>
```

<key> references a key specified in a directory's user tag

<user> represents the value of the key

<domain> is the domain to which the user belongs

Example:

```
user_exists id 1000 192.168.1.101
```

will return true where there exists in the directory a user with a key called id whose value equals 1000:

User Directory Entry

```
    <user id="1000" randomvar="45">
        <params>
          <param name="password" value="1234"/>
          <param name="vm-password" value="1000"/>
        </params>
        <variables>
          <variable name="accountcode" value="1000"/>
          <variable name="user_context" value="default"/>
          <variable name="effective_caller_id_name" value="Extension 1000"/>
          <variable name="effective_caller_id_number" value="1000"/>
        </variables>
    </user>
```

In the above example, we also could have tested for randomvar:

```
user_exists randomvar 45 192.168.1.101
```

And we would have received the same true result, but:

```
user_exists accountcode 1000 192.168.1.101
```

or

```
user_exists vm-password 1000 192.168.1.101
```

Would have returned false.

See Also

- Channel Variables